Implementación de un generador TRNG en FPGA

Alumno: Gonzalo Castro Castilla

Tutor: Enrique San Millán

Agradecimientos

Me gustaría aprovechar esta oportunidad para agradecerles a todos aquellos que me han ayudado y apoyado durante mis años en la universidad en general y estos últimos meses en particular.

En primer lugar agradecer la ayuda que me ha prestado Enrique San Millán, tutor de este trabajo, así como la comprensión y disponibilidad que me ha dado durante estos meses.

Agradecer también la comprensión que han mostrado tanto mis compañeros de laboratorio como mis jefes de Prosegur. Su actitud flexible me ha facilitado mucho el poder finalizar de forma exitosa el proyecto.

También me gustaría darles las gracias a todos mis amigos de la universidad. Un agradecimiento especial a Jon Goitia por enseñarme a entregar las prácticas "bien formateadas".

Agradecer a mis padres el gran esfuerzo que han realizado para que pueda estudiar en la universidad Carlos III de Madrid y por su apoyo y cariño. También agradecer a mi hermano mayor por ser siempre un modelo para mí.

Finalmente, agradecer a Belén por la paciencia que ha tenido conmigo, su amor y su apoyo. Sin ella no habría podido ser tan feliz todos estos meses.

Muchas gracias.

Gonzalo Castro

Tabla de contenidos

Índice de Figuras

Índice de Tablas

1. Introducción y objetivos

La aleatoriedad juega un papel muy importante en la actualidad en campos como la seguridad, las telecomunicaciones y las matemáticas. Esto unido al avance de la informática y la electrónica ha hecho imprescindible el desarrollo de sistemas capaces de generar números aleatorios.

Esta necesidad unida a la gran flexibilidad que ofrecen sistemas como las FPGAs en el desarrollo de sistemas digitales hace de este proyecto una buena oportunidad para estudiar los generadores de números aleatorios en hardware y el cambio en su comportamiento al modificar ciertos parámetros.

Es importante señalar que, debido al carácter determinista, es decir, predecible, de los sistemas digitales, la implementación de generadores de números aleatorios es un proceso más complejo de lo que podría parecer *a priori*.

A continuación se introducirán los conceptos de aleatoriedad y entropía, así como los métodos más comunes para la generación de números aleatorios en sistemas digitales y algunos conceptos clave que se emplearán a lo largo del proyecto.

Posteriormente se desarrollarán los objetivos propuestos para el proyecto, así como los diferentes pasos intermedios que se plantean antes de alcanzarlos.

1.1. Números aleatorios

El matemático Derrick Henry Lehmer dijo en 1951, "Una secuencia aleatoria es un concepto poco claro en que cada término es imprevisible para los no iniciados y cuyas cifras pasan un cierto número de pruebas tradicionales con los estadísticos" [1].

Existe una idea generalizada e intuitiva sobre la aleatoriedad. Por ejemplo si se tuviese una secuencia binaria de ceros y unos "1010101010" la primera impresión

es que no se corresponde a una secuencia aleatoria, sino que es una secuencia de ceros y unos predecible, siendo los próximos valores “1” y “0”.

Si ahora la secuencia cambiase y fuese “0100100001” resultaría más complicado encontrar el patrón que siguen los ceros y los unos, por lo que podríamos llegar a pensar que se trata, efectivamente, de una secuencia de números aleatorios.

La realidad es que si cogemos una moneda y asociamos la cruz y la cara al cero y al uno respectivamente, las probabilidades de obtener una secuencia u otra son exactamente iguales, 2^{10} [2]. Debido a que las probabilidades de que obtengamos ambas secuencias son las mismas y ambas están generadas por el mismo suceso -el lanzamiento de monedas- debemos considerar que las dos secuencias son, realmente, secuencias aleatorias de ceros y unos.

Este análisis nos obliga a rechazar la interpretación de la aleatoriedad como la ausencia de patrones y nos obliga a buscar otra definición del concepto. También nos obliga rechazar la idea de poder juzgar si un solo número es o no aleatorio en sí mismo [3], sino que su aleatoriedad vendrá marcada dependiendo de que el proceso que la ha generado sea o no aleatorio [2].

El concepto de aleatoriedad y su estudio tuvo un fuerte avance a mediados del siglo XX, durante la segunda Guerra Mundial. Esto se debe a que la aleatoriedad juega un papel fundamental en la ciencia de las telecomunicaciones.

Para ilustrar la relación de los números aleatorios con las telecomunicaciones, pensemos en el mensaje de antes: “1010101010”. Si se desease enviar este mensaje podría hacerse de dos maneras. La primera sería transmitir literalmente los 10 bits de los que está compuesto. La segunda opción sería transmitir la información resumida en un algoritmo capaz de generar esa secuencia, como “imprimir ‘10’ cinco veces”.

De esta forma si en vez de tener 10 bits, tuviésemos que enviar 100 millones de bits con la misma secuencia, la segunda opción sería claramente más eficiente.

Si ahora quisiésemos enviar un segundo mensaje con el código "0100100001", solo podríamos realizar el envío de los bits literalmente. En este caso no podríamos encontrar un algoritmo sencillo que nos permitiera obtener esta secuencia de bits ocupando menos que el mensaje original.

Para poder medir las diferencias entre cada secuencia en términos de compresibilidad, se usa la llamada "complejidad de Kolmogórov". Ésta se fija en la cantidad de recursos necesarios a nivel computacional para describir una cantidad de información. Por eso podemos llegar a la conclusión de que en ocasiones el mensaje será mínimamente comprimible (si la complejidad de describirlo es tal que merece la pena transmitirlo tal cual) o directamente imposible, al ser aleatorio. Podríamos decir que una serie de datos responde al azar cuando el algoritmo ocupa más cantidad de información para describirlo que la secuencia en sí.

Otro de los grandes campos de conocimiento que aprovecha en gran medida la aleatoriedad es la criptografía. Los mayores avances en este ámbito los podemos observar desde comienzos del siglo XX. En las últimas etapas de la Primera Guerra Mundial, el ejército alemán (Wehrmacht) comenzó a desarrollar un sistema nuevo, concretado en una máquina llamada Enigma

Esta máquina empleaba una posición inicial aleatoria, de la que dependía el cifrado del mensaje completo, logrando así dificultar el espionaje en sus comunicaciones secretas [4].

La relación entre aleatoriedad y cifrados es importantísima, siendo uno de los motivos por los que se busca la implementación de buenos generadores de números aleatorios y por los que su estudio y comprensión se ha vuelto clave en el mundo de la criptografía y de las telecomunicaciones.

Podemos encontrar multitud de otros usos de la aleatoriedad o su apariencia: apuestas y juegos de azar, loterías, muestreos estadísticos, mercado de valores bursátiles, arte, simulaciones por ordenador, etc. Siguiendo el razonamiento expuesto, la verdadera aleatoriedad cobra especial relevancia en el área de la criptografía, uso para el cual se centrará este documento

1.2. Aleatoriedad estadística y secuencias aleatorias algorítmicas

Hasta ahora se ha analizado cómo el concepto de aleatoriedad difiere del concepto de "ausencia de patrones reconocibles", sin embargo no se ha dado una definición completa del concepto. Conviene aclarar primero otros términos cercanos:

1.2.1. Proceso estocástico

Proviene del griego "hábil en conjeturar". Es el sistema que se comporta de forma no determinista. Un proceso es estocástico, por tanto, si no podemos conocer el estado concreto en el que estará el sistema, incluso si el conjunto de estados posibles es conocido de antemano. No siendo predecible, decimos que es aleatorio. Sin embargo, puede estudiarse desde el punto de vista probabilístico.

Por ejemplo, el experimento de lanzar una moneda n veces nos da un 50% de probabilidades de obtener cada resultado. Sobre el estado concreto del sistema, podemos afirmar la parte de información conocida: que será o bien cara o bien cruz. Sin embargo, cada suceso es independiente y no sabemos qué orden de resultados tendremos a lo largo del experimento. La secuencia no sigue un patrón conocido pese a que el experimento sigue una distribución de probabilidad conocida (Bernoulli).

Incluso si se descubriese una mayor probabilidad hacia uno de los resultados del experimento -por ejemplo, que en la cara el relieve de la moneda utilizase más

material y tendiese por ello a estabilizarse en el aire colocándose en la posición inferior-, seríamos incapaces de describir la secuencia de resultados del experimento.

Por ello, es importante tener en cuenta que a pesar de conocer en mayor profundidad la fuente de aleatoriedad, no tiene por qué significar que el experimento sea determinista, sino que seguiría siendo aleatorio, como se demostrará en el capítulo dedicado a los tests más adelante.

Como puede observarse, la ausencia de patrones reconocibles es condición necesaria pero no suficiente para afirmar la presencia de aleatoriedad. Es necesario además la falta de previsibilidad que experimentos como el de la moneda aportan. Por eso podemos considerar que éstos son una buena fuente de aleatoriedad, mientras que otras fuentes que tampoco tienen un patrón de resultados en la secuencia, no lo son. Es el caso del número π. Sus decimales no incluyen la repetición de un grupo de números (no existe patrón detectable), pero sí podemos calcularlos, son previsibles. Además, cada número o grupo de números se corresponde con una posición concreta dentro de la secuencia, de modo que, sabiendo un extra de información, en este caso, sí ayuda a la predicción de los siguientes, puesto que no se trata de sucesos aleatorios independientes.(5).

1.2.2. Teoría de la información

Es la rama de conocimiento matemática e informática que estudia cada elemento que interviene en la comunicación: canales, mensajes, códigos, compresión, criptografía... [5]

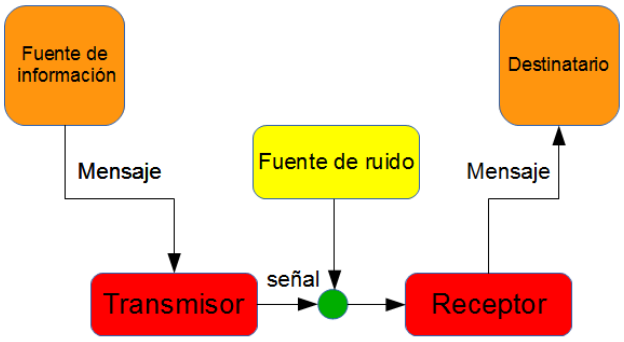


Ilustración 1 - Elementos de la teoría de la información

La teoría de la información persigue la mejor forma de transmitir mensajes codificados con rapidez, seguridad y economía de recursos [6]. El mensaje, que consiste en el caso digital, en un conjunto de unos y ceros, porta la información. Así pues, la información es proporcional cuantitativamente a la cantidad de bits que se requieren como mínimo para representar el mensaje. La probabilidad de repetición de un mensaje determina la cantidad de información. Es decir, la redundancia afecta al valor que le otorguemos. Así pues, la aleatoriedad implica más información, mientras que los trozos repetidos, aportan menos cantidad de ella.

Por ejemplo, si un estudiante de la Universidad Carlos III escribiese a la secretaría de alumnos y su mensaje se titulase "Convalidación de asignaturas cursadas en la Universidad Carlos...." es altamente probable que la continuación de los caracteres que faltan, sea "III". El valor de esos caracteres sería muy bajo en principio. Sin embargo, si en realidad el final del mensaje fuese "Albizu", la información en realidad contenía un valor mucho más alto. Debido a que casi todos los alumnos que escriben a la secretaría terminan el mensaje "Carlos" con "III", la redundancia es alta. Si en lugar de eso, se tratase de la terminación del nombre de una asignatura, la variabilidad sería mucho más amplia, aunque todavía podríamos tomar un número finito de resultados predecibles y ordenarlos en función de su probabilidad.

El código puede facilitar la información, aumentando el valor de cada partícula informativa y haciendo más eficiente su transmisión. Por ejemplo, asignando un número a cada asignatura. Si tanto el emisor como el receptor conocen el código y lo utilizan adecuadamente, el sistema mejora. El código que permite resumir la información es útil para la compresión y transmisión de datos.

Pero el código también puede ocultar la información: por ejemplo, en el caso de firmar con las iniciales J.M., en lugar de con el nombre completo, el alumno introduciría la incertidumbre entre todas las combinaciones, más o menos probables de la información verdadera (José Manuel, Juan Manuel, José María, Jefferson Mohammed...). Además, podría inducir a más incertidumbre en otros datos, como podría ser si lo hiciese para evitar identificarse como hombre o mujer (Juana María).

Cualquier mensaje lleva un código asociado: desde el concepto de números binarios o decimales hasta el idioma en el que se expresa. Todos ellos asumen una información externa, adicional al propio mensaje. Si se sabe que "presunto" está expresado en castellano, tendrá un significado distinto a si conocemos que está dicho en portugués -significando "jamón"-. Si sabemos el código, podemos interpretar el mensaje. Si no lo sabemos, pero podemos averiguarlo basándonos en las probabilidades de construcción del mensaje, es posible acabar consiguiéndolo igualmente. Por ejemplo, empezando por las combinaciones de letras más comunes en el idioma en cuestión. Sin embargo, si no contamos con esas distintas probabilidades de ocurrencia, al introducir la aleatoriedad y homogeneizar la probabilidad de los posibles resultados, podremos transmitir información de forma segura, sin riesgos de que llegue aunque el mensaje llegue a las manos equivocadas, éste obtenga de él la información.

La criptografía, por tanto, necesita aprovechar la "desinformación", y de hecho, generarla. Por el contrario, los métodos de "desencriptación" tienen como objetivo reducir la incertidumbre de la información, y por eso buscan en muchos

casos secuencias con alta probabilidad de repetición. Basándose en la redundancia, el que intenta romper un sistema de seguridad probará primero con lo más probable (por ejemplo, en castellano, repitiendo más las vocales a y e que la u o la i) y sólo si no tiene éxito, comenzará a usar otros métodos. La criptografía trata de dificultar la predictibilidad de la información igualando la probabilidad de cada suceso.

1.2.3. Entropía de la información o Entropía de Shannon

Mide la incertidumbre de una fuente de información. Es el grado de "desinformación" que tiene un conjunto de datos para "tener sentido". Sigue los conceptos ya mencionados de probabilidad de cada fragmento del mensaje para establecer valores en una distribución [6]. Si cada uno de ellos es igualmente probable, la distribución es plana, por lo que la incertidumbre y por tanto la entropía es máxima [7]. La aleatoriedad, por tanto, genera la máxima entropía de la información posible.

1.3. Generación de números aleatorios

Los generadores de números aleatorios se dividen en dos tipos según la fuente de entropía que usen [8], [9]. En particular podemos diferenciarlos según sean:

- PRNG o DRNG (Pseudo random/deterministic number generator): su fuente de aleatoriedad yace en un número aleatorio llamado semilla. Posteriormente un proceso determinista se encargará de generar números usando dicha semilla.
- TRNG (True random number generators): su fuente de aleatoriedad se basa en un proceso físico analógico (ruido térmico, ruido eléctrico, decaimiento radioactivo, etc.). A diferencia de los PRNGs no se guarda ningún estado en el generador y la salida solo se basa en dicho proceso físico.

Debido a que los PRNG siguen un proceso determinista, su aleatoriedad queda limitada a la aleatoriedad de la semilla proporcionada. Generalmente, la semilla de los PRNG suele ser generada por un TRNG y se suele sustituir cada poco tiempo para evitar periodicidades. En definitiva, un PRNG es inseguro en sí mismo si no dispone de un TRNG que proporcione la fuente de aleatoriedad necesaria [3].Es por ello que los TRNG merecen tal grado de atención, ya que son un componente clave en la calidad de todo sistema de seguridad [10].

Ya que los TRNG, a diferencia de los PRNG, no son deterministas, su eficiencia, entendida como la calidad de la aleatoriedad que generan, puede ser menor que la de los PRNG si no están bien equilibrados [11]. Haciendo un símil con una moneda, un TRNG mal equilibrado se corresponde a una moneda trucada en la que es más probable un cierto resultado que otro.

Por tanto, dependiendo de la función que vayan a desempeñar, será recomendable el uso de un tipo u otro o la combinación de ambos. A continuación se presenta una tabla a modo de comparativa entre los TRNG y los PRNG, así como de la recomendación de usar uno u otro dependiendo de la aplicación para la que se les vaya a emplear.

Característica	PRNG	TRNG
Eficiencia	Excelente	Pobre
Determinista	Sí	No
Periodicidad	Periódico	No periódico
Predecible	Sí	No

Tabla 1 - Comparativa de PRNGs y TRNGs

Estas características harán que dependiendo de la aplicación se recomiende el uso de un PRNG o TRNG:

Aplicación	Tipo de generador recomendado
Loterías	TRNG
Juegos y azar	TRNG
Muestreos aleatorios	TRNG
Simulación y modelado	PRNG
Seguridad (p.ej. generadores para cifrado)	TRNG
Artes (p.ej. dibujos aleatorios)	Ambos

Tabla 2 - Generador recomendado para diferentes aplicaciones

Como se ha comentado previamente, es posible que generen unos bits con menor aleatoriedad que los PRNGs. Esto significa que, en algunas ocasiones, los bits generados, a pesar de ser aleatorios, tienen una tendencia hacia uno de los dos valores posibles. En estos casos es posible realizar algún tipo de reducción de tendencia, como se explicará en el apartado 2.4, aunque no es algo imprescindible.

En el caso de introducir circuitos de reducción de tendencias se deberá vigilar que estos circuitos introducidos no generen correlaciones, ya que al intentar reducir la tendencia podríamos estar provocando números periódicos, destruyendo, así, la fuente de entropía [12]

1.4. Objetivos

El objetivo de este proyecto es el de implementar un TRNG siguiendo el modelo propuesto por Kohlbrenner y Lockheed [13]. Este diseño hace extensible la arquitectura de TRNGs en FPGAs que no dispongan de PLLs.

Los PLLs, o bucles de enganche de fase, son unos dispositivos que permiten generar osciladores con muy buenas características para construir un TRNG. Tienen un período constante, y permiten un ajuste muy preciso de su frecuencia de oscilación.

No obstante, las FPGAs que disponen de PLLs tienen un coste superior al de aquellas que no los tienen. Este diseño plantea la posibilidad de sustituir los PLLs por unos bloques lógicos programables presentes en todas la FPGAs: los CLBs (Configurable Logic Blocks).

Los objetivos que se plantean en el proyecto, serán, por tanto:

1. Construir un TRNG según el modelo propuesto y analizarlo para optimizarlo:
    - Estudio del comportamiento de los osciladores, frecuencia y estabilidad, variando diferentes parámetros.
    - Estudio de la variación del rendimiento del generador en función de diferentes parámetros.
    - Mejorar el rendimiento del generador usando para ello los resultados adquiridos en los puntos anteriores.

2. Comparar los resultados obtenidos con otros generadores de tipo PRNG.

2. Estado del arte

En esta sección se profundizará en los conceptos introducidos anteriormente y en su aplicación directa en el presente trabajo. Además, se mostrará una perspectiva de la situación actual sobre los TRNGs.

También se hará referencia a las distintas alternativas de diseño de un generador de números aleatorios en hardware, así como de técnicas de mejora de resultados.

2.1. Los TRNG en la actualidad

El diseño e implementación de máquinas capaces de generar números aleatorios verdaderos se ha perseguido desde hace tiempo. Uno de los primeros ejemplos de máquinas capaces de generar números aleatorios verdaderos fue patentado en el año 1946 por la compañía estadounidense de telecomunicaciones ATT. La máquina generaba números aleatorios mediante bolas de colores negro y blanco para el cifrado de las telecomunicaciones [14].

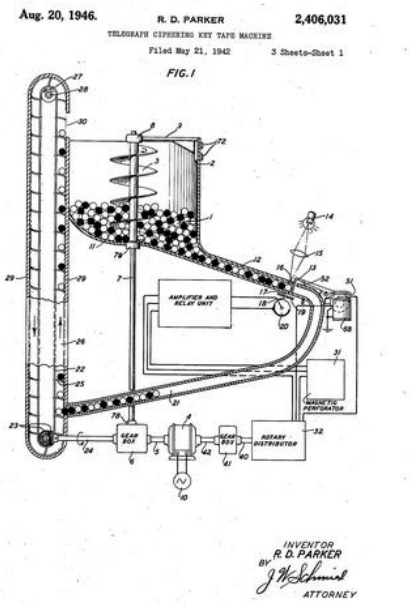


Ilustración 2 - Generador de números aleatorios de ATT, año 1946 [14].

Evidentemente, estos sistemas eran muy costosos y voluminosos. Además, al ser fácilmente observables es fácil romper la seguridad, ya que al poder conocer la secuencia aleatoria de semilla, el cifrado está comprometido. Por ello lo lógico era remplazar estos sistemas con sistemas eléctricos y electrónicos, que son más fiables, discretos y compactos.

Sin embargo, y debido a la naturaleza determinista de los sistemas digitales, la generación de números aleatorios en estos sistemas no es algo trivial. Un ejemplo de esto es el hecho de que hasta hace relativamente poco en los sistemas operativos más extendidos solo se incluían instrucciones que generaban números aleatorios mediante secuencias deterministas.

En la actualidad los TRNGs comienzan a incluirse en los procesadores, actuando siempre como un complemento –para proporcionar la semilla- a los PRNGs. A finales del año 2012 Intel anunció la inclusión de una nueva instrucción en sus procesadores, RDSEED, que permitiese la obtención de números realmente aleatorios. No obstante, la misma compañía recomienda su uso solo para proporcionar la semilla a su PRNG, invocado mediante la instrucción RDRAND, ya que genera números de mayor calidad que la instrucción RDSEED [15]. En la siguiente tabla se comparan ambas instrucciones y sus propósitos:

Instrucción	Fuente	Resultado NIST
RDRAND	PRNG seguro para cifrados	SP 800-90A
RSEED	Generador de bits aleatorios no determinista	SP 800-90B & C (drafts)

Tabla 3 - Comparación entre RDRAND y RDSEED

En la actualidad hay una amplia gama de generadores de números aleatorios mediante hardware en el mercado. Se pueden encontrar generadores desde 100$ hasta 6000$, lo que nos da una idea de que realmente son sistemas con una gran aplicación.

2.2. Alternativas de diseño: Arquitecturas del TRNG

Existen diferentes posibilidades a la hora de implementar un TRNG mediante un circuito digital. Los métodos más comunes son lo que emplean osciladores con topología de anillo. Dentro de éstos se distinguen dos grupos principales: los que emplean una pequeña variación en la frecuencia de estos osciladores -denominada jitter- y los que usan la metaestabilidad como fuente de entropía. [10]

Aunque se ha optado por los TRNG de topología de anillo que emplean la variación en la frecuencia por tener un uso más extendido [13], ambos conceptos –el de metaestabilidad y el de jitter- son muy relevantes en el proyecto, por lo que se desarrollarán en capítulos posteriores.

2.3. Osciladores de anillo y "jitter"

Como ya se ha comentado en el apartado 1.4, el objetivo del proyecto es implementar y estudiar el rendimiento de un generador en FPGAs que solo disponga de CLBs. Para ello se usarán unos componentes que se denominan osciladores de anillo o "ring oscillators".

Un oscilador en forma de anillo es un circuito no-estable, es decir, un circuito que que posee dos estados "cuasi-estables" entre los que conmuta. Los osciladores resultan una forma sencilla y efectiva de construir TRNGs debido a que su frecuencia se ve afectada por diferentes parámetros [16].

El oscilador en forma de anillo más sencillo se puede implementar colocando en forma de cascada un número impar N de inversores conectados en forma de cascada formado un bucle cerrado. [17].

De forma ideal, la señal de salida será una señal cuadrada periódica Ψ, cuyo período dependerá del número de inversores y el retraso de los mismos. Esto es [18]:

$$T = n \cdot \tau \text{ y } \Psi(t) = \Psi(t + T)$$

Donde T es el período y τ es el retraso de un solo inversor.

Sin embargo, la señal de salida no es una señal cuadrada perfecta, sino que su período varía de una forma aleatoria según [12]:

$$T = T + T\hat{}$$

Donde T^ representa una variable aleatoria que toma valores en el rango (−T /2, T /2).

En un oscilador de alta calidad el rango de T^ deberá ser muy pequeño comparado con el período de la señal. Esta vibración en la señal de reloj, T^, se suele denominar "jitter" y es, precisamente éste el que se usará como fuente de entropía.

En la siguiente figura se representa mediante diferentes líneas el concepto de "jitter" en la señal del oscilador:

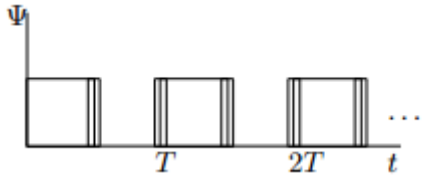


Ilustración 3 - Representación gráfica del "jitter"

La idea detrás de un TRNG que utilice la arquitectura de dos osciladores comentada previamente, es la de extraer, en cierta manera, este "jitter" o imprecisión en la frecuencia y que, de alguna forma, pase a representar estados binarios finitos, es decir ceros y unos.

2.4. Técnicas de mejora de resultados

Como ya se ha visto, los TRNGs generan números aleatorios sin seguir procesos deterministas, por lo que suelen producir tendencias y que la probabilidad de obtener un "0" o un "1" no sea del 50%. Esto no implica que el generador deje de ser aleatorio, pero sí que su rendimiento se verá afectado en ciertos tests que comparen la cantidad de "0" y "1" en grandes secuencias.

Por ello, y para mejorar el rendimiento de los generadores se han desarrollado diferentes técnicas para reducir esta tendencia. Todas ellas se basan en el análisis posterior de los datos obtenidos. Es importante recalcar que estas estrategias no son necesarias en todos los diseños, pero sí son recomendables.

Una de las primeras técnicas desarrolladas fue la propuesta por el matemático húngaro Von Neumann y se basa en la conversión de parejas de bits según la siguiente tabla:

Bits consecutivos	Salida
0, 0	No hay resultado
0, 1	1
1, 0	0
1, 1	No hay resultado

Tabla 4 - Técnica de mejora de aleatoriedad de Von Neumann

Por ejemplo, si se tuviese un generador con un 0.6 de probabilidad para el "0" y un 0.4 para el "1" y tendríamos esta secuencia de partida:

"01 00 01 11 10 01 10 10 11 11"

Tras procesarla siguiendo la propuesta de Von Neumann obtendríamos:

"1 N 1 N 0 1 0 0 N N"

Así se puede ver cómo se mejoraría la proporción de los dos tipos de bits. No obstante hemos rechazado entorno a un 50% de los bits generados, lo que nos lleva a la conclusión de que este tipo de compensación tiene un coste en la tasa de bits por segundo.

Otra alternativa más compleja es la del uso de una función de tipo "hash" como el SHA-1 [19]. Este tipo de funciones generan otros números a partir de los bits proporcionados, de forma que un pequeño cambio en el número que se le introduzca en la función generará una salida completamente diferente a otro número muy cercano o similar.

El uso de este tipo de métodos se debe hacer con cuidado ya que éstos pueden incluir patrones y periodicidades que eliminarían el factor de aleatoriedad y la fuente de entropía.

3. Desarrollo del proyecto

Esta sección del trabajo se compone de dos partes principales. La primera consiste en el desarrollo teórico: se detalla, explica y desarrolla el diseño propuesto por Kohlbrenner y Lockheed [13]. Además incluye las dificultades técnicas encontradas durante su diseño, las simulaciones, la construcción y finalmente la síntesis del mismo.

Por otro lado, en la segunda parte, se presentan los resultados obtenidos de los experimentos realizados con el propósito de estudiar cómo varía la respuesta del circuito a diferentes cambios en su implementación o en su diseño.

3.1. Desarrollo teórico

3.1.1. Descripción general del generador

El TRNG se compone de 4 bloques principales:

- Dos osciladores de anillo, usados como fuente de entropía (CLK0 y CLK1)
- Un muestreador.
- El módulo de transmisiones RS-232 para la comunicación con el PC.

Estos bloques se colocan y relacionan de la siguiente manera:

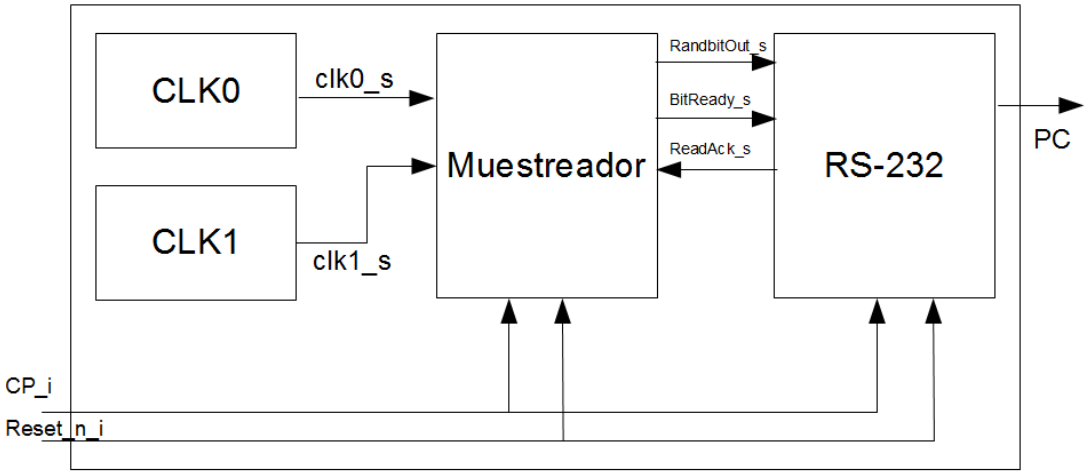


Ilustración 4 - Disposición general de los bloques del diseño

Siendo:

- CP_i: el reloj de la FPGA – usando el modelo Spartan3e 50 MHz-
- Reset_n_i: el reset asíncrono para resetear todas la variables de estado y los flip-flops

El funcionamiento general es el siguiente: CLK0 y CLK1 generarán unas señales con una cierta frecuencia f. Muestreando una señal con respecto a la otra y si ambas tienen una frecuencia similar, habrá un momento en el que una muestree a la otra durante el jitter. Esto dará como resultado una señal, S0_s, de frecuencia variable. Esa variación en la frecuencia es aleatoria y será, por tanto la fuente de aleatoriedad del circuito.

Una vez muestreada la señal, el resto del circuito del muestreador se encarga de extraer y detectar cuándo hay un nuevo bit aleatorio listo. Esto se indica mediante la señal "BitReady_s", que cambiará a "1" si existe un bit nuevo disponible. Entonces el nuevo bit, "RandBitOut_s", quedará registrado en el módulo RS-232 y esto se le indica al muestreador mediante la señal "ReadAck_s".

Una vez tengamos 8 bits -con los que se forma un byte- éstos se transmitirán al PC mediante el puerto RS-232. Durante ese tiempo no se muestreará nada. Una vez finalizada la transmisión se repetirá el proceso anteriormente descrito.

A su vez, mediante un programa, corriendo en el PC, se hará la lectura de los datos así como el formateo y escritura en un fichero para su posterior análisis en el PC.

En los puntos posteriores se hará una descripción más detallada del funcionamiento de cada uno de los módulos enumerados previamente. Posteriormente se desarrollan los problemas que se han tenido durante la implementación de dichos módulos.

3.1.2. Diseño del oscilador

3.1.2.1. Funcionalidad y diseño

El diseño del generador de números aleatorios consta de dos osciladores. Estos osciladores son unos circuitos combinacionales realimentados de forma que no tengan un comportamiento estable.

Un oscilador se puede implementar de diferentes formas, siendo los más empleados los que utilizan puertas de tipo NOT, LUTs y buffers. En nuestro diseño se ha decidido utilizar unos osciladores compuestos únicamente de puertas NOT y una puerta AND que activa o desactiva dicho oscilador.

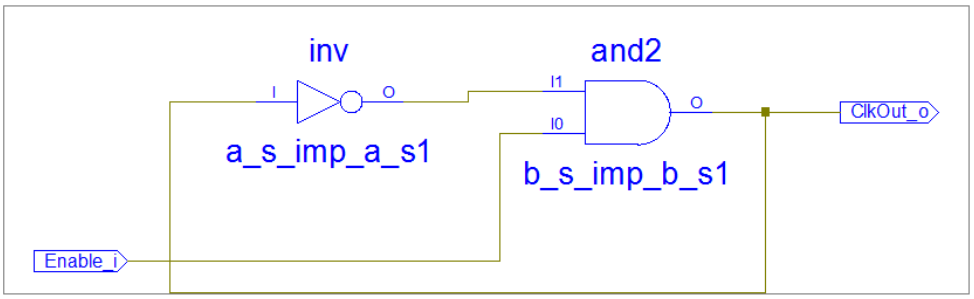


Ilustración 5 - Oscilador básico (una NOT y una AND)

Su comportamiento es el siguiente:

1. Si la señal de Enable_i es '0', entonces la señal "b" valdrá '0' independientemente del valor de la señal "a".
2. Si Enable_i es '1', entonces la señal "b" pasará a valer el valor contrario de "a", es decir, NOT(A). Justo después, "a" pasará a valer NOT(A), alterando su valor inicial.

Este circuito se comportará siempre de esta forma si que se usa un número de puertas NOT impar. Así, la salida del oscilador, CLKOut_o, nos dará una señal que

variará con una frecuencia f. Dicha frecuencia es el resultado de la acumulación de los retrasos de cada puerta, p_i, de forma que [10]:

$$f = \frac{1}{2\sum p_i}$$

Como consecuencia, al poner más puertas obtendremos una frecuencia más baja, ya que los diferentes retardos en la transmisión de la señal se irán acumulando, aumentando el denominador de la fórmula. Esto se comprobará de forma empírica en el análisis de resultados propuestos en el capítulo 4 en .

Hemos de ser conscientes de que aunque una FPGA es un circuito digital, la estamos empleando de una forma poco convencional al generar señales de una frecuencia superior al reloj del sistema. Por ello, no obtendremos una señal de onda cuadrada, sino algo más parecido a una señal sinusoidal. Esto ocurre porque las señales no pueden cambiar de forma instantánea de valor, debido a las capacidades parásitas de los transistores con los que están hechos los sistemas digitales actuales.

3.1.2.2. Simulación Post-place and Route y resultados reales:

Podemos imaginarnos que la simulación de este tipo de circuito no podrá realizarse con un banco de pruebas convencional. Esto se debe a que este circuito no tiene un sentido lógico si, como suponen las herramientas de simulación, los retardos no existen y las señales se transmiten de manera instantánea.

Una de las formas de estimar el tiempo de retardo entre nuestras puertas es realizar una simulación "Post place and route". Este tipo de simulación se puede hacer una vez sintetizado el diseño. Como indica su nombre, requiere como paso previo la colocación y el rutado de las puertas. Así, proporcionándole al software de simulación el fichero de los retardos entre cada puerta, se llegará a ver una simulación de los osciladores.

De esta forma, por ejemplo, al tener un oscilador con cinco puertas NOT y una AND como la de la imagen y realizar una simulación "Post place and route", obtenemos una frecuencia de unos 94 MHz:

$$91610\ ps - 81008\ ps = 10602\ ps$$

$$T = 10602\ ps \rightarrow f = 94\ MHz$$

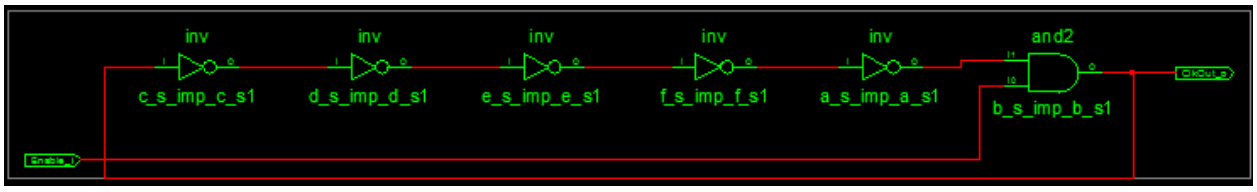


Ilustración 6 - Oscilador implementado con 5 NOTs y una AND

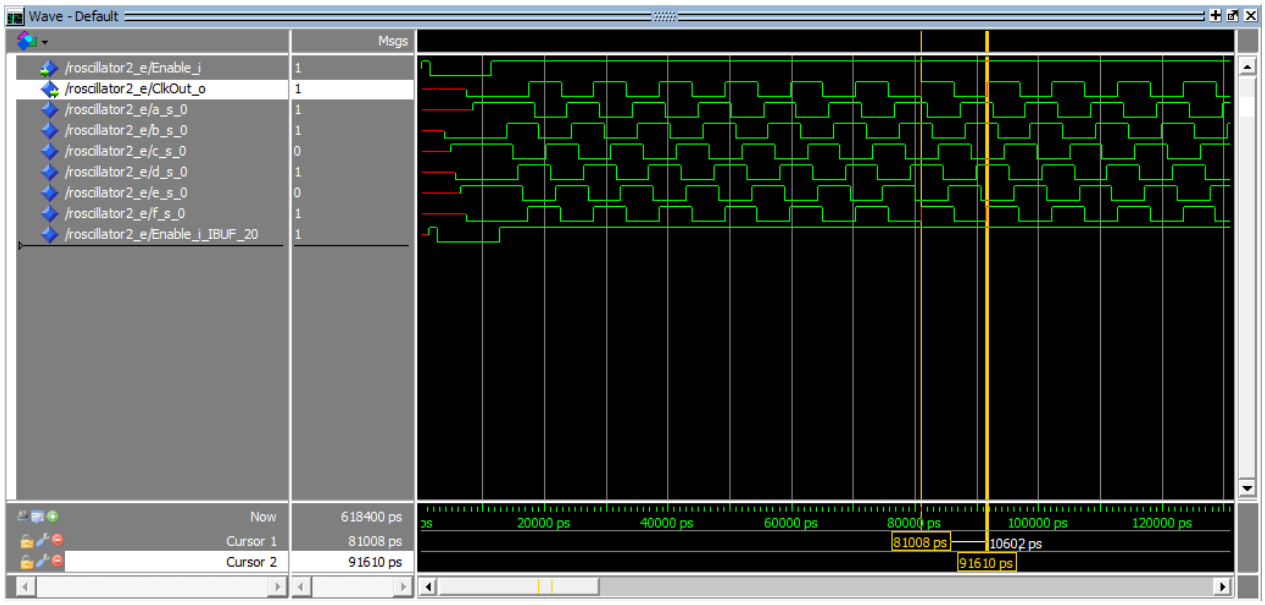


Ilustración 7 - Simulación post place and route del oscilador

Al realizar el paso a la FPGA el resultado obtenido es similar al de la simulación, pero como se ha comentado previamente, la señal no es una onda cuadrada, sino más bien un seno. Esto se debe a que tan pronto como la señal cambia

de valor, volverá a invertirse sin mantener la señal en el 0 o el 1 lógico durante mucho tiempo.

Como se puede apreciar, la señal adquiere valores negativos. Esto es un comportamiento extraño en sistemas digitales. En este caso vuelve a ser consecuencia del diseño inusual que tiene el oscilador comparado a la forma normal de desarrollo en FPGAs.

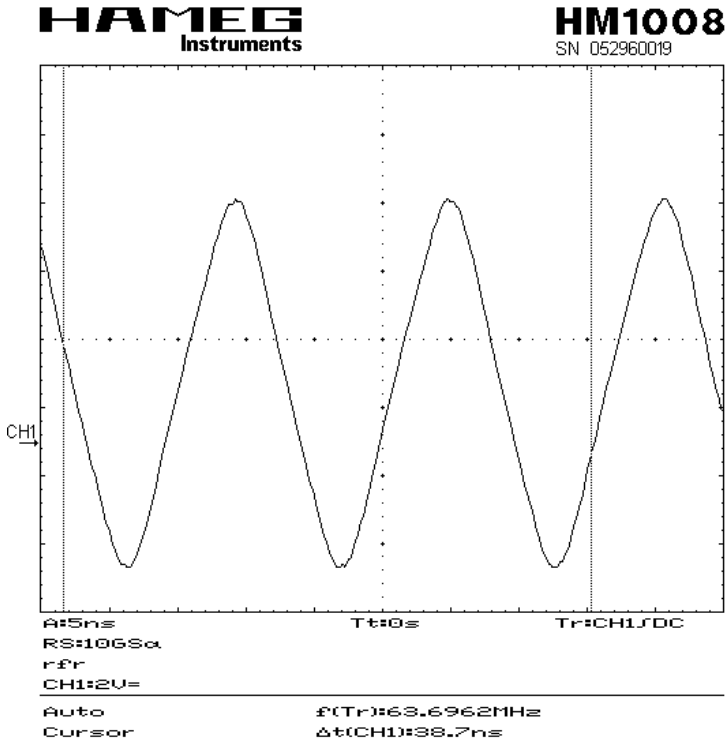


Ilustración 8 - Imagen real generada por el oscilador

Como se puede observar, la frecuencia obtenida se encuentra en el orden de magnitud aproximado de la frecuencia predicha por el simulador en la simulación PPNR. Sin embargo, muchas de las características de la señal no se muestran en la simulación. Por ejemplo, nuestra señal, que en la realidad tiene un período cambiante de entre 98 y 120 MHz no muestra este comportamiento en la simulación. Esto tiene dos consecuencias:

1. El sistema completo no es simulable, ya que el período cambiante de los osciladores no se puede simular mediante una simulación PPNR.
2. Realmente tenemos una fuente de aleatoriedad, ya que si el sistema completo fuese simulable, sería un sistema determinista.

Además del número de puertas existentes, el período de la señal de salida puede verse afectado por la colocación de las puertas lógicas. Esto puede explicarse recordando que las señales tendrán que recorrer más espacio hasta llegar de una puerta a otra y por tanto el retardo en la señal será mayor.

Otro de los factores que pueden influir es la temperatura. Aunque la temperatura a la que está la FPGA es un factor complicado de controlar sabemos que con un aumento de la temperatura, se produce un aumento de la conductividad del material semiconductor y una disminución en la conductividad de los metales. Debido a que en una FPGA se presentan ambos compuestos es difícil prever su efecto.

Todos estos factores serán objeto de estudio en el apartado destinado a los resultados experimentales, mostrando los resultados obtenidos en las diferentes señales así como sus frecuencias.

3.1.3. Diseño del muestreador o "sampler"

3.1.3.1. Funcionalidad y diseño

El muestreador es el circuito encargado de extraer la entropía y transformarla en bits. Para ello hace uso de una parte asíncrona, encargada de la extracción del jitter, y otra parte síncrona encargada de actuar como mecanismo de control sobre la parte asíncrona.

En la siguiente ilustración se muestra el diseño del muestreador señalando las partes síncronas y las asíncronas.

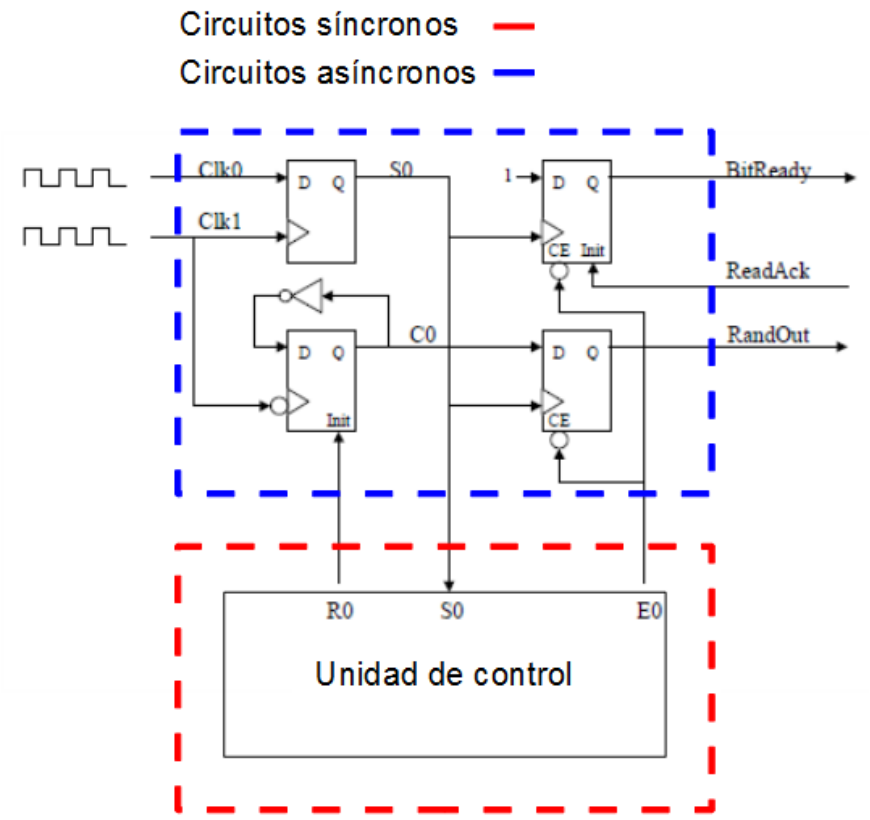


Ilustración 9 - Diseño del muestreador. Imagen modificada de [13]

Su funcionamiento es el siguiente: clk1_s muestrea a clk0_s mediante un flip-flop tipo D. Ésta señal se denomina S0_s. La idea detrás de este sistema es que

Clk0_s y Clk1_s tengan una frecuencia muy similar y estable. Así, cuando haya un flanco ascendente de S0_s, es decir, que clk1_s ha pasado a muestrear los unos de clk0_s, se genere un número aleatorio.

Este número será aleatorio porque una pequeña variación en la frecuencia de los osciladores hará que la frecuencia de S0_s cambie. Estos pequeños cambios en la frecuencia hacen que sea totalmente impredecible saber si el bit aleatorio generado será un "0" o un "1". Finalmente, estos bits aleatorios son el resultado del muestreo de C0_s por parte de S0_s.

En la siguiente ilustración se muestran las señales críticas del sistema y cómo se generan los bits aleatorios mediante un pequeño cambio en la frecuencia de clk0_s, es decir, su jitter.

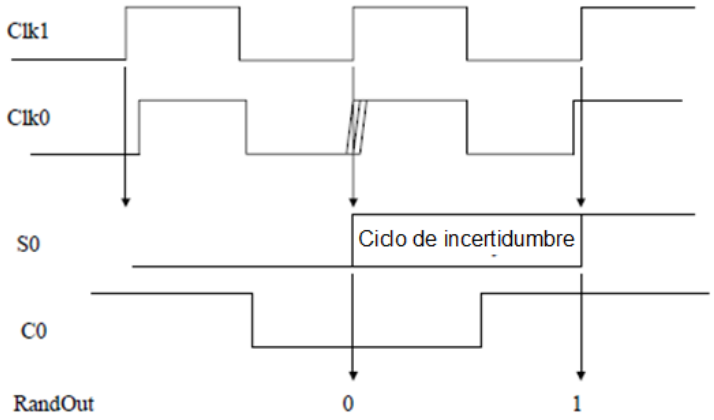


Ilustración 10 - Señales críticas del muestreador. Modificado del original obtenido de [13]

3.1.3.2. Unidad de control

Ya se ha visto que estas señales, a pesar de que en teoría deberían ser cuadradas, tienen un comportamiento realmente analógico. Por eso es necesario tener en cuenta que puede haber oscilaciones no deseadas que activen ciertos flip-flops.

Como consecuencia, hay que tener en cuenta este tipo de comportamientos y diseñar mecanismos para evitar que estos picos sean interpretados como señales y no como ruido, su verdadera naturaleza.

En la imagen a continuación se muestra cómo una de estas oscilaciones en S0_s puede hacer creer al módulo de comunicaciones UART que hay un bit listo (BitReady) cuando en realidad no es así.

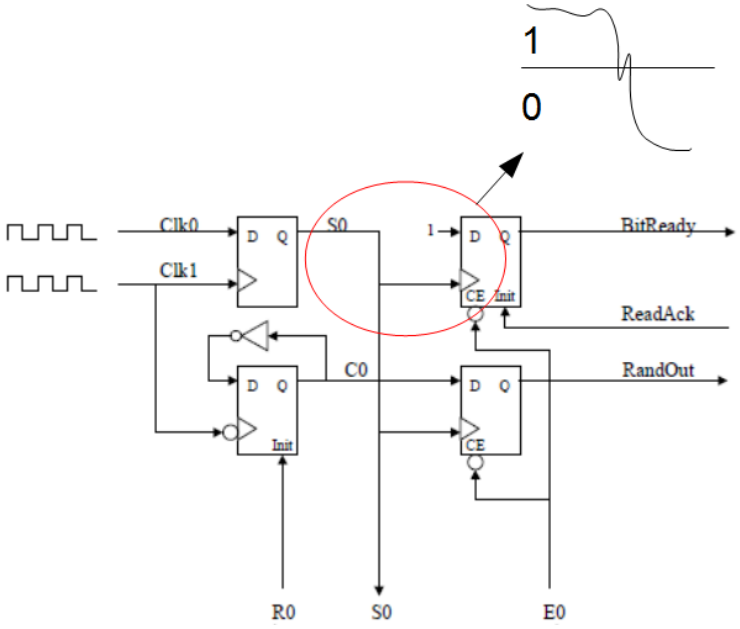


Ilustración 11 - Posibles oscilaciones en S0_s. Original obtenido de [13]

Para evitar este efecto, un circuito de control se encarga de desactivar E0 una vez se haya generado un bit aleatorio, controlando así los flip-flops causantes de BitReady y RandOut según un tiempo estipulado en un Preset.

Este Preset debe ser un valor bajo para que cumpla su función: evitar estas oscilaciones en S0_s. Si se le da un valor más alto –entorno al número 10 en decimal- lo único que ocurrirá es que se rechazarán más valores y se tardará más tiempo en tener un bit aleatorio disponible.

3.1.4. Diseño del módulo RS-232

Para el análisis de los datos obtenidos es necesario pasarlos a un ordenador, de forma que se le puedan aplicar los tests de aleatoriedad.

La Spartan 3E, modelo de FPGA usada en este proyecto, dispone de un puerto USB. Sin embargo no se puede usar para efectuar comunicaciones con la FPGA, ya que es sólo un JTAG para su programación y comprobación de errores.

Una vez descartada la opción de usar el USB de la FPGA la alternativa es usar el puerto serie. El gran inconveniente de este puerto es su lentitud en la transmisión. Por otro lado, su facilidad de uso y utilización hace que sea un recurso muy útil en este trabajo.

3.1.4.1. El protocolo serie

El puerto serie es un tipo de comunicación en el que se transfiere un bit cada vez. Cada bit se envía a una velocidad fija, que tiene que conocer previamente tanto el emisor como el receptor. En la actualidad, la frecuencias disponibles son: 75, 110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 y 115200 bit/s.

Aparte de la tasa de transmisión, hay otros parámetros que deben ser conocidos por ambos -emisor y receptor- para la correcta transmisión de los datos. Estos son:

- El tipo de saludo o "Handshake": hay secuencias especiales de bits que están reservadas para iniciar la comunicación y para cerrarla. También puede prescindirse del saludo e iniciar la comunicación directamente
- Bits de paridad: Una forma de comprobar que la comunicación se ha realizado correctamente y no ha habido alteraciones en la misma es asignar un bit de paridad. El valor de este bit dependerá directamente del resto de byte transmitido. De esta forma se comprueba si el bit de paridad realmente se corresponde con la secuencia que se ha recibido.

El resto de la comunicación se realiza siempre de la misma manera, siguiendo la siguiente estructura:

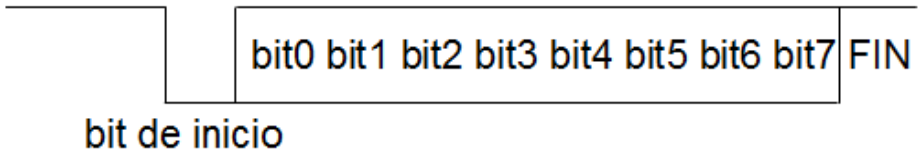


Ilustración 12 - Estructura del protocolo serie

Cuando el sistema está en reposo se transmite un nivel alto. Esto es un sistema heredado de las antiguas comunicaciones telefónicas en las que esto indicaba que la infraestructura está operativa.

Para iniciar la comunicación se envía un pulso bajo y posteriormente los 8 bits. Para terminar, se transmitirá el bit final, siempre un nivel alto.

3.1.4.2. Limitaciones del puerto serie

Incluso la mayor de las tasas de transferencia, 115200 baudios (pulsos por segundos), es demasiado baja para la comunicación entre la FPGA y el PC. Debido a ello, la comunicación serie es el factor limitante del tiempo que tarda la FPGA en transmitir todos los bits necesarios al ordenador.

Los tests requieren como mínimo 100 millones de bits aproximadamente. Si se transmiten 115200 bits cada segundo en total, generar cada fichero tardará:

$$\frac{100.000.000}{115200} \cdot \frac{1}{60} = 14{,}47\ minutos$$

Esto es un aspecto a mejorar en el proyecto y, como alternativa se propone la posibilidad de realizar los tests directamente en la FPGA.

3.1.4.3. Funcionalidad y diseño

La función básica del módulo RS-232 es almacenar bits según se vayan generando. Para ello, cuando llega el primer bit, generado por el muestreador, el módulo RS-232 lo guarda en un flip-flop e informa al muestreador de que ha sido guardado activando el bit ReadAck_s.

Una vez el bit ReadAck_s se ha activado, la señal que indica que existe un bit disponible se pone a "0" hasta que vuelva a generarse un nuevo bit.

Este proceso se repite ocho veces, hasta que se hayan guardado un total de 8 bits. En ese momento se detiene el proceso de almacenaje de bits y se comienza a transmitirlos. Para ello, se ha implementado un divisor de frecuencia que, a partir de los 50 MHz del reloj de la FPGA, genera una señal que se activa 115200 veces por segundo. Como se ha mencionado anteriormente, esta es la mayor de las frecuencias posibles para el puerto serie. La señal rxOtx_s es la encargada de indicar si el módulo RS-232 se encuentra en el estado de envío (activa) o de recepción de datos (cero).

El módulo de comunicaciones serie se ha instanciado mediante una máquina de estados implícita. Esto significa que no se ha declarado una variable de tipo estado que almacene los posibles estados en lo que puede estar el sistema.

Además, el sistema dispone de un reset asíncrono controlable de forma externa. En caso de que el reset esté activo se detendrá el envío de números a través del puerto serie.

3.1.4.4. Simulación y banco de pruebas

La simulación del módulo RS-232 se ha realizado junto a la del muestreador, ya que ambos necesitan comunicarse entre sí. Podemos ver cómo la imagen de la simulación se asemeja a la obtenida mediante ChipScope, más abajo:

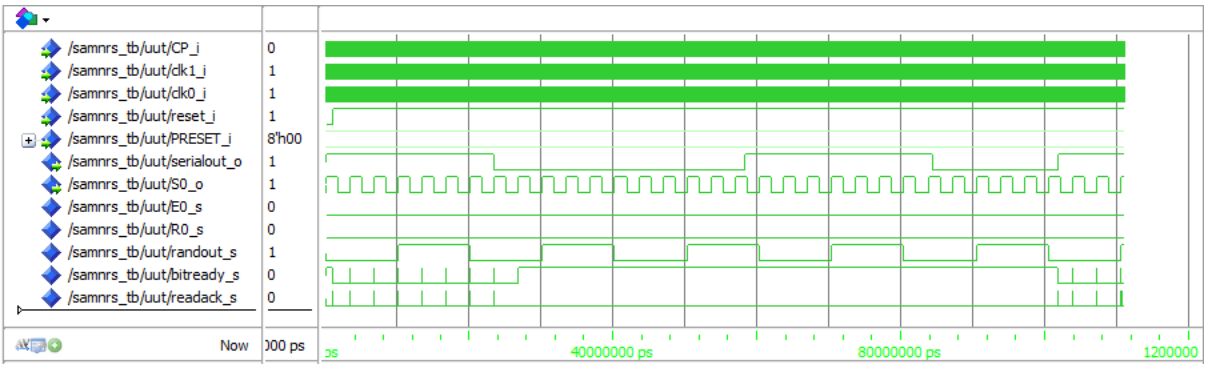


Ilustración 13 - Simulación de los módulos muestreador y RS-232

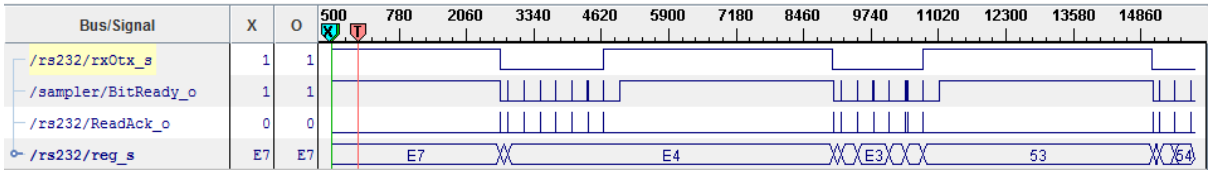


Ilustración 14 - Señales de comunicación serie obtenidas mediante ChipScope

La mayor de las diferencias entre la simulación y la realidad es aquella en la que la simulación para rx0tx_s es "0", pero en ReadAck adquiere un valor positivo de forma periódica. Sin embargo, en ChipScope, se puede ver cómo ReadAck no se pone a "1" de forma periódica.

3.1.5. Consideraciones sobre señales asíncronas

De forma general, en cualquier diseño en FPGAs, el diseño asíncrono de sistemas debe evitarse [20]. Los circuitos asíncronos son sistemas que no se rigen por los estímulos del reloj del sistema, CP_i, por lo que pueden cambiar de estado en cualquier momento. A continuación se exponen los motivos de porqué este tipo de diseño no es recomendado y se explica cómo se ha solventado este problema.

El diseño implementado tiene una parte asíncrona, que actúa como fuente de aleatoriedad y está formada por los dos osciladores de anillo y parte del muestreador. Por otro lado, las comunicaciones serie con el PC y parte del muestreador son circuitos síncronos, que se rigen por el estímulo del reloj del sistema.

En la siguiente ilustración se diferencian las partes del circuito que son síncronas y las asíncronas:

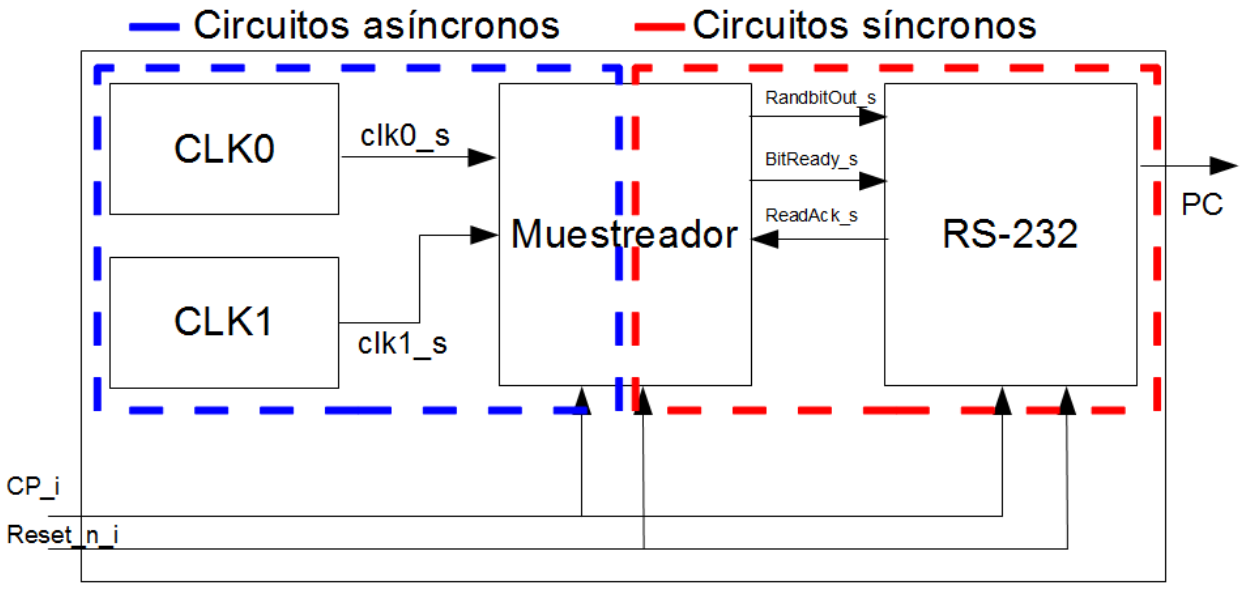


Ilustración 15 - Circuitos síncronos y asíncronos del diseño

Estos elementos asíncronos son imprescindibles como fuente de aleatoriedad. Lo habitual en el diseño normal de sistemas digitales es evitar su uso, precisamente porque introducen comportamientos impredecibles en el sistema.

Para ser consciente del problema que puede suponer una entrada asíncrona se debe pensar en qué ocurre si dicha señal cambia de estado justo cuando haya un flanco ascendente de reloj. En este caso, puede suceder que mientras unos flip-flops han recibido el flanco ascendente de reloj, a otros aún no ha llegado (recordemos que las señales tardan un tiempo en transmitirse). Entonces se produce un estado "imposible" en el que unos flip-flops han recibido el cambio de la señal asíncrona y otros en el que no, por lo que mantienen un estado anterior.

En nuestro caso, esto se ve claramente en el módulo RS-232, síncrono, cuyas entradas Bit_ready_s y RandBit_s son asíncronas. A continuación se presentan unas imágenes que muestran qué sucede si obviamos este comportamiento singular de las FPGAs. Estas imágenes se corresponden a los primeros instantes de ejecución después de desactivar el reset:

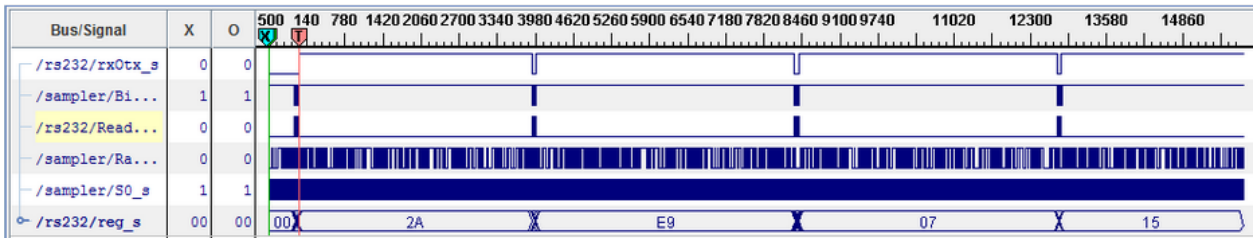


Ilustración 16 - Imagen de Chipscope: Error en la ejecución (Inicio)

Como ya se ha explicado en el punto 3.1.1, el módulo RS-232 tiene dos estados alternativos. Las posibilidades son:

1. Enviar (rx0tx_s = 1).
2. Guardar hasta un máximo de 8 bits (rx0tx_s = 0).

Si nos fijamos en el punto 8325 más detalladamente, podemos ver cómo el módulo, en lugar de guardar 8 bits, memoriza 9 bits. Esto no es posible y es un comportamiento no contemplado en el diseño de este trabajo:

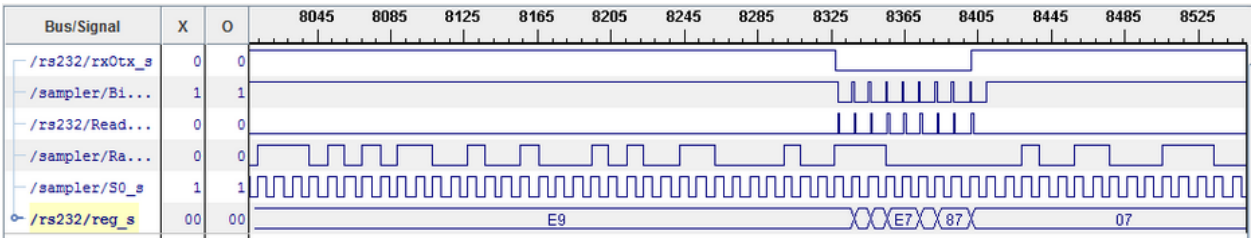


Ilustración 17 - Imagen de Chipscope: Error en la ejecución (Detalle)

Además, unos instantes después, tras haber transmitido entre 10 y 12 caracteres, el módulo RS232 deja directamente de transmitir nada, comportándose de una forma muy extraña, ya que no se está almacenando ningún bit. Conviene fijarse ahora en que los flancos de bajada son mucho más estrechos en esta imagen que en la primera, utilizando ambas la misma escala:

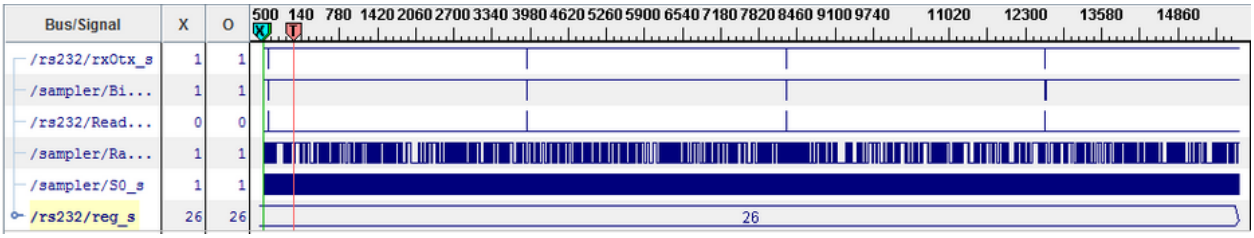


Ilustración 18 - Imagen de Chipscope: Error en la ejecución (1 min después del inicio)

Estas imágenes se corresponden al valor real de las señales, obtenido mediante el software ChipScope. No se trata de una simulación.

Este tipo de comportamiento no es apreciable en una simulación, donde funcionará sin ningún tipo de problema, ya sea una simulación normal, o una "Post place and route". Además, como el propósito del generador construido es contar con una fuente de números aleatorios, resulta imposible generar los estímulos con las mismas características que en la forma en la que se producen en la realidad.

Debido a que este tipo de errores no se aprecian en las simulaciones, se debe evitar en la medida de lo posible el diseño asíncrono de los sistemas.

Para resolver este problema, la solución más sencilla es colocar un flip-flop a la entrada de la señal asíncrona. Esto hará que la señal de entrada quede sincronizada mediante el reloj.

Bajo estas circunstancias, el flip-flop de sincronización no está en un estado definido, que podría asemejarse al equilibrio inestable en mecánica, llamado metaestabilidad. En teoría, un flip-flop podría mantenerse en este estado de metaestabilidad, sin embargo las perturbaciones térmicas y las asimetrías en los retardos de las señales hacen que el flip-flop "caiga" hacia uno de los lados.

Para ilustrarlo podríamos pensar en una pelota que está justo en la parte más alta de un tejado y que caerá con la más mínima ráfaga de aire:

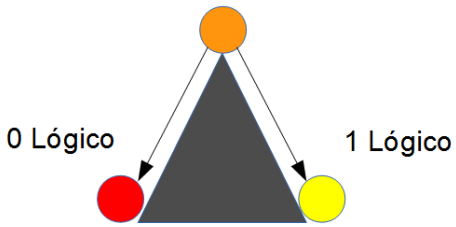


Ilustración 19 - Metaestabilidad

Existen tres alternativas para evitar el problema de la metaestabilidad [21]:

- Aumentar el período del reloj del sistema.
- Colocar dos o más sincronizadores (flip-flops) en serie, de forma que la probabilidad de que los dos a la vez estén en metaestabilidad sea muy baja [22].
- Usar una estrategia de temporizado (timing strategy) independiente de la velocidad de los circuitos individuales.

La opción elegida ha sido la segunda, ya que el reloj no es modificable y la tercera opción, a pesar de ser interesante, es más compleja que la segunda.

Una vez colocados los dos flip-flops el módulo RS-232 funciona correctamente y no se dan situaciones no previstas como al inicio. Podría darse el caso de que aun con dos flip-flops no funcionase. Entonces sería necesario añadir un tercero para que disminuya la probabilidad de que los tres se encuentren en estado de metaestabilidad a la vez.

En la siguiente imagen se muestra el funcionamiento correcto del sistema, también con ChipScope. Ya se puede apreciar cómo Read_Ack_o se pone solo 8 veces al valor "1" cada vez que rxOtx_s adquiere el valor "0":

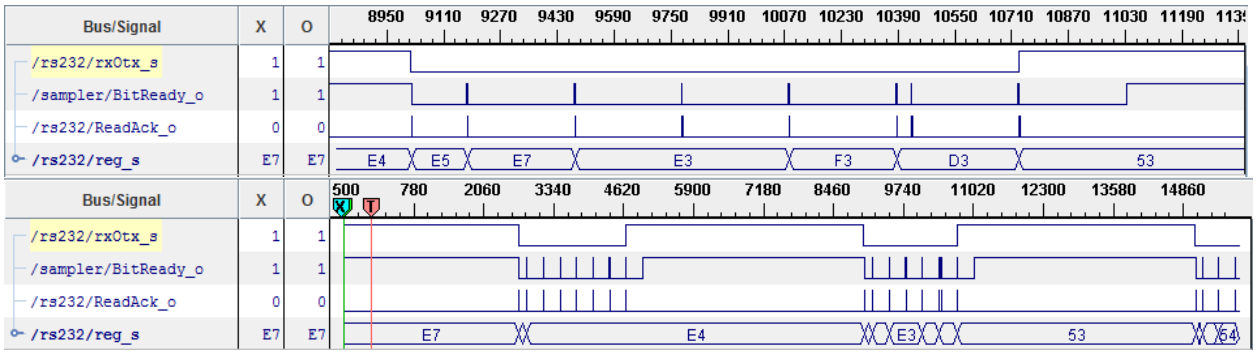


Ilustración 20 - Funcionamiento correcto en ChipScope después de eliminar la metaestabilidad

3.1.6. Hard Macro vs. Relatively Placed Macro

Para facilitar el análisis de los osciladores en particular y de todos los circuitos críticos en general, podemos usar dos formas de diseño muy útiles en este caso: las "hard macro" y las "relatively placed macro". Estas estrategias permiten controlar el orden y la colocación de los diferentes elementos, permitiéndonos hacer estudios sobre el cambio del rendimiento según la colocación de los elementos.

Las Hard-Macro son una forma de diseño que nos permite generar unas "cajas negras" cuyos rutados y posiciones relativas quedan definidas. Estas "cajas negras" son reutilizables en otros lugares de la FPGA. Esto tiene dos ventajas muy notables:

- ahorrar tiempo a la hora de hacer el rutado de las pistas, ya que está realizado
- poder ocultar el contenido de dicha "caja negra" al usuario, permitiendo así poder distribuir el circuito ya implementado sin dar a conocer su diseño.

Por otro lado, las Relatively Placed Macro –aunque son un concepto similar a las Hard-Macro- son otra solución para simplificar el realizar diferentes instanciaciones de la misma entidad.

Las Relatively placed macro representan una alternativa más flexible que las Hard-macro, ya que su instanciación y definición es más flexible y versátil que las de las Hard –Macro. Por este motivo son las elegidas para el probar diferentes distribuciones de los osciladores.

Según Xilinx [23] una RPM es una colección de elementos (FFS, LUT, CY4, RAM, etc.) que se colocan en un grupo llamado "set". La colocación de cada elemento es la clave para crear RPMs y la colocación de los elementos dentro del set

en relación a otros elementos se controla mediante Relative Location Constraints (RLOCs).

La definición de las RPMs se realiza en el archivo de restricciones (.ucf). Un ejemplo de la definición del oscilador "clk0" se puede ver en la siguiente imagen.

```
INST "clk0/a_s1" U_SET=myrpm;
INST "clk0/b_s1" U_SET=myrpm;
INST "clk0/c_s1" U_SET=myrpm;
INST "clk0/d_s1" U_SET=myrpm;
INST "clk0/e_s1" U_SET=myrpm;
INST "clk0/f_s1" U_SET=myrpm;
```

```
INST "clk0/a_s1" RLOC= X0Y0;
INST "clk0/b_s1" RLOC= X0Y1;
INST "clk0/c_s1" RLOC= X0Y2;
INST "clk0/d_s1" RLOC= X0Y3;
INST "clk0/e_s1" RLOC= X0Y4;
INST "clk0/f_s1" RLOC= X0Y5;
```

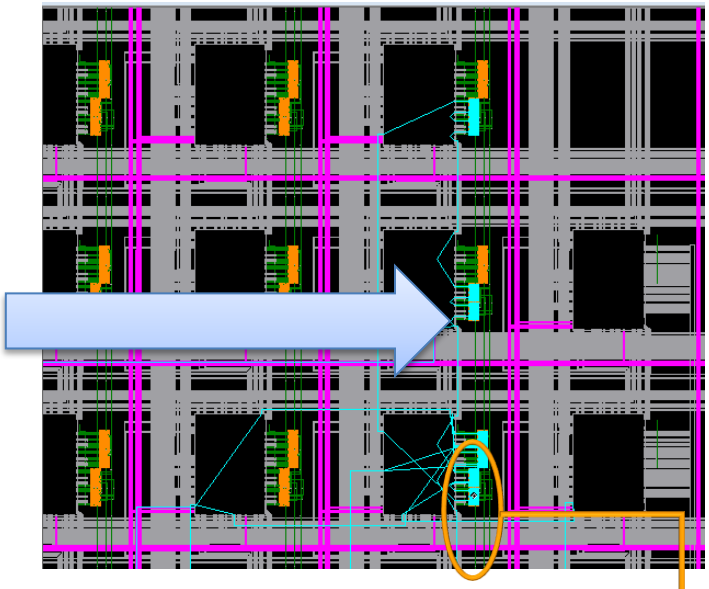


Ilustración 21 - Ejemplo de Relatively Placed Macro

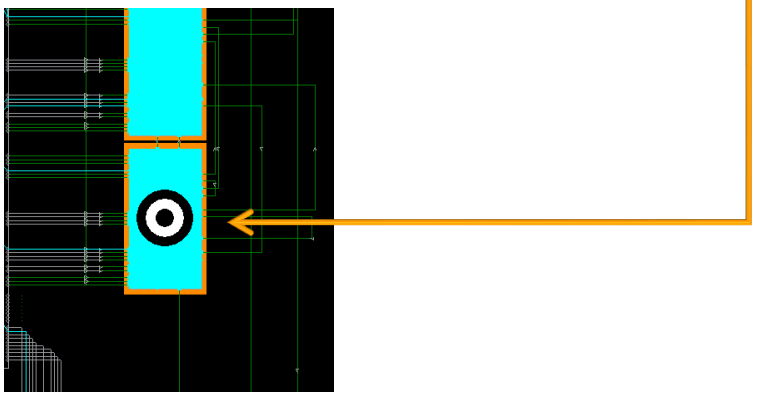


Ilustración 22 - Origen de la Relatively Placed Macro generada

Mediante la instrucción "U_SET" se crea un grupo, denominado en este caso, "myrpm" y mediante RLOC se hace una colocación relativa dentro del grupo.

3.1.7. Evitar la eliminación de señales

Las herramientas disponibles actualmente para la síntesis de circuitos optimizan los circuitos programados. Esto, en la mayoría de los casos, es beneficioso, ya que mejora el rendimiento del circuito y le proporcionan velocidad y estabilidad. No obstante, en algunos casos se podría desear que no se eliminaran señales y evitar así esta simplificación

Esto es lo que sucede con los osciladores de anillo que se desean instanciar. Si se quiere sintetizar un oscilador como el de la Ilustración 23 la herramienta detectará que hay señales redundantes desde el punto de vista de la lógica y se eliminarán. Esto es, si la señal "A" la invertimos dos veces, obtendremos una señal, "C" que tendrá el mismo valor que "A".

Desde el punto de vista digital y de la lógica ambas señales, "A" y "C" serían exactamente idénticas. Sin embargo, desde el punto de vista de nuestra aplicación estas señales no son idénticas, ya que existe un retraso entre cada una de las señales.

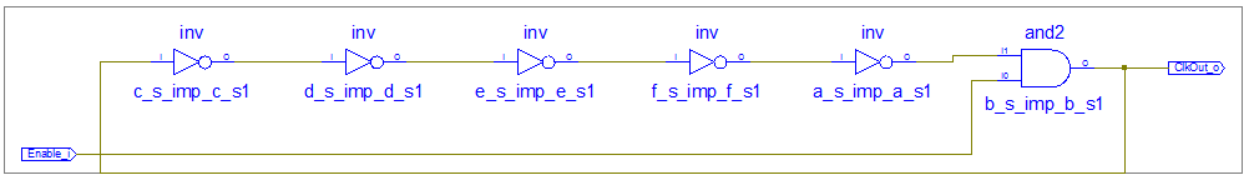


Ilustración 23 - Oscilador sin simplificar

Debido a esta simplificación se obtendrá un circuito como el de la Ilustración 24, ya que el sintetizado eliminará todas las puertas NOT que encuentre:

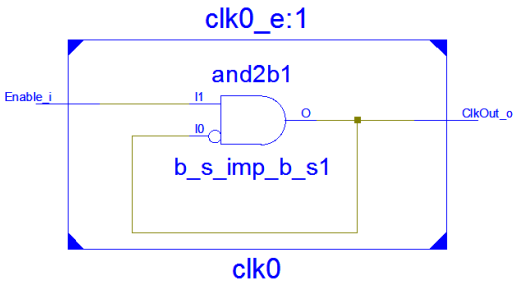


Ilustración 24 - Oscilador simplificado en la síntesis

Para evitar que las señales desaparezcan, se deben incluir unas instrucciones en la arquitectura de los osciladores indicando qué señales se deben mantener. Estas instrucciones serán de la siguiente manera para un oscilador con cinco puertas NOT:

```
attribute keep: boolean;
attribute keep of a_s: signal is true;
attribute keep of b_s: signal is true;
attribute keep of c_s: signal is true;
attribute keep of d_s: signal is true;
attribute keep of e_s: signal is true;
attribute keep of f_s: signal is true;
```

Una vez se haya colocado este código en el fichero de restricciones ya no se simplificarán ni eliminarán las señales indicadas. Por otro lado, la herramienta de síntesis seguirá avisándonos de que el diseño utilizado no es óptimo ya que tiene bucles lógicos.

```
WARNING:Xst:2170 - Unit top_e : the following signal(s) form a combinatorial loop
```

Ilustración 25 - Aviso del sintetizador: loop combinacional

Esto no debería ser motivo de preocupación, ya que es precisamente lo que se deseaba al incluir estas instrucciones en el fichero de restricciones.

3.1.8. Consideraciones sobre S0_s, clk0_s y C0_s

En el diseño propuesto hay tres señales que son de vital importancia en el proceso de generación de los números. Estas señales son: S0_s, clk0_s, clk1_s y C0_s.

Como ya se ha comentado, la fuente de aleatoriedad del generador surge en los pequeños cambios que hay en la frecuencia en las señales cuadradas periódicas clk0_s y clk1_s, producida por los osciladores de anillo. Estas señales tienen una frecuencia que ronda los 100 MHz y una pequeña variación en su frecuencia.

Clk_1 muestrea a clk0_s y, como resultado, se obtiene otra señal, denominada S0_s. Esta señal, de menor frecuencia que clk0_s y frecuencia variable será la encargada a su vez de muestrear otra señal, C0_s, que no es más que clk0_s cuya frecuencia reducida a la mitad. En el siguiente gráfico se pueden apreciar las 4 señales así como imágenes obtenidas mediante el osciloscopio:

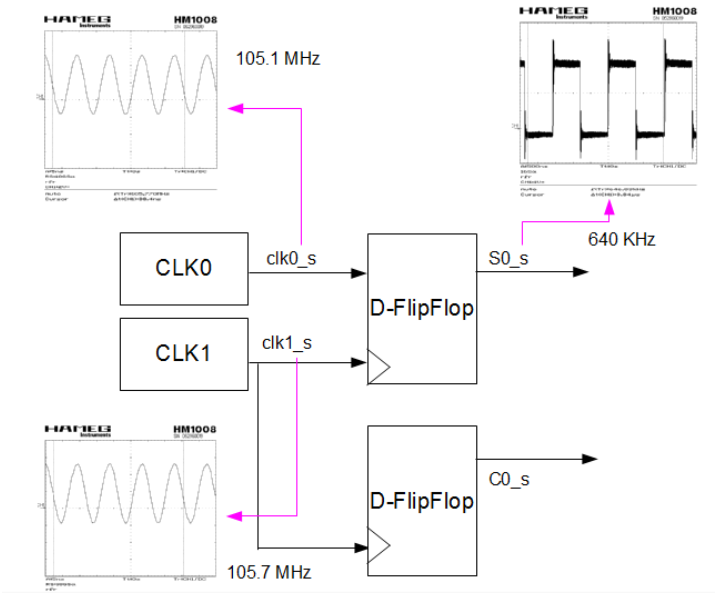


Ilustración 26 - Resumen de las señales críticas (I)

Estas señales son las denominadas señales críticas, ya que su forma y frecuencia varían totalmente dependiendo de factores como su posición, el número de puertas NOT con las que estén diseñadaso la temperatura a la que se encuentren.

Esto se debe a que el diseño no es síncrono y son señales que tienen una alta dependencia en los retrasos de las puertas lógicas de la FPGA.

En la siguiente ilustración se puede apreciar cómo al cambiar la cantidad de puertas NOT de los que están compuestos los osciladores cambia la frecuencia de todas las señales, siendo, desde el punto de vista lógico, el mismo circuito:

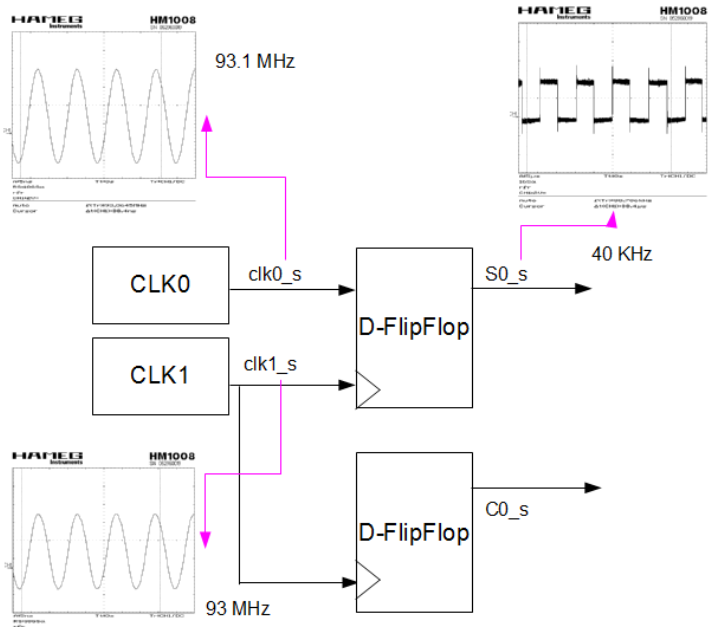


Ilustración 27 - Resumen de las señales críticas (II)

De entre todas las señales hay una que merece especial atención. Esa señal es S0_s, resultado del muestreo de clk0_s por parte de clk1_s. Tiene una gran importancia ya que nos proporciona una idea de cómo son las señales de los osciladores y nos informa de su frecuencia y la diferencia entre una y otra.

La frecuencia de una señal resultante (f_a) de muestrear una señal (f_{in}) con respecto a otra (f_s) es:

$$f_a(N) = |f_{in} - Nf_s|$$

Donde N es un número entero.

Para entender un poco mejor la fórmula dada, se incluye la siguiente tabla. Se supone que clk1_s queda fija a 100 MHz:

Frecuencia de entrada	N	Frecuencia de salida (S0_s)
90 MHz	0	100-90 = 10 MHz
99.9 MHz	0	100-99.9 = 100 KHz
230 MHz	1	230-100*2= 30 MHz

Ilustración 28 - Frecuencia de alias en señales muestreadas

Ya que el fin de este proyecto es implementar un generador basado en jitter, se intentará conseguir dos propósitos:

1. La frecuencia de los osciladores debe ser muy parecida.
2. Los osciladores deben tener una frecuencia lo más estable posible.

Como la frecuencia de los osciladores debe ser similar, N siempre será 0, por lo que la frecuencia que se obtiene en S0_s es directamente la diferencia entre las señales de entrada y la frecuencia de muestreo. Esto es:

$$f_a = |f_{in} - f_s|$$

Ya que la frecuencia de los osciladores suele ser muy alta, resulta difícil medir las diferencias entre sus períodos. Se usará S0_s para medir cómo de parecidas son las señales de los osciladores. Así una baja frecuencia de S0_s nos indicará que ambas señales son muy similares entre sí, mientras que una alta frecuencia nos indicará que las señales son muy diferentes.

En principio, lo que nos interesa es tener una S0_s baja, para asegurarnos de que seguimos el principio de funcionamiento planteado. Sin embargo, ya veremos como una S0_s baja no nos garantiza la aleatoriedad del sistema.

Una alta S0_s alta, además de por una diferencia en la frecuencia, puede estar también producida por una gran inestabilidad de la señal. Por ejemplo, un oscilador compuesto por 5 puertas NOT y una puerta AND. Cada puerta tendrá un retraso asociado de unos nanosegundos, sin embargo, este retraso no es fijo sino que tiene una incertidumbre asociada y puede variar. Esto hará que la señal de salida del oscilador no tenga un período bajo.

Se ha comprobado que la frecuencia de S0_s tiende a bajar a medida que baja la frecuencia de los osciladores. Esto se debe en gran medida a que las señales más rápidas se ven mucho más afectadas por el ruido, y por tanto son mucho más inestables que las señales lentas, a las que les da tiempo a llegar a estabilizarse.

3.1.9. Lectura y formateo de los datos

La lectura de los datos se puede realizar de diferentes formas. Existen programas como Putty que permiten configurar todos los parámetros de la conexión serie de una forma fácil e intuitiva.

En la siguiente imagen se muestra la configuración requerida para leer los datos desde el puerto serie número 4 que es en el que se sitúa el puerto del ordenador.

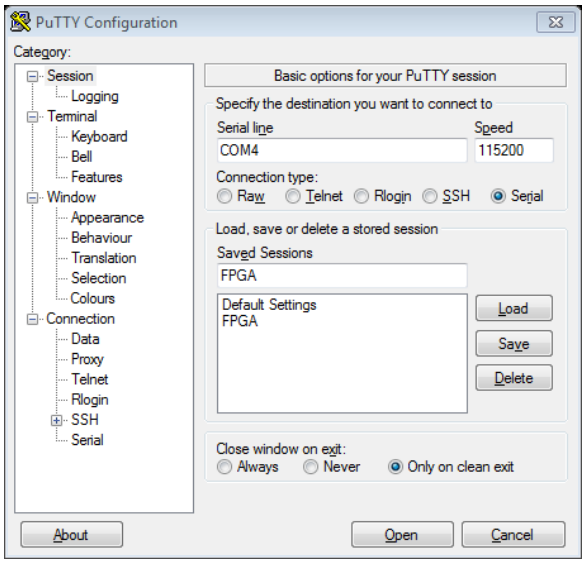


Ilustración 29 - Configuración de Putty

Aunque Putty ofrece una opción rápida y sencilla de comprobar si se están recibiendo datos por el puerto serie, no permite una manipulación compleja de los mismos.

Por este motivo se ha optado por usar un lenguaje de programación de alto nivel, es decir, de alto grado de abstracción del hardware, que permita una configuración rápida y sencilla de la lectura, así como una manipulación cómoda de los mismos.

Se ha optado por Python por su sencillez, claridad en el código, versatilidad y existencia de una librería dedicada exclusivamente a la lectura de datos por el puerto serie. A continuación se incluye el diagrama de flujo del programa en Python:

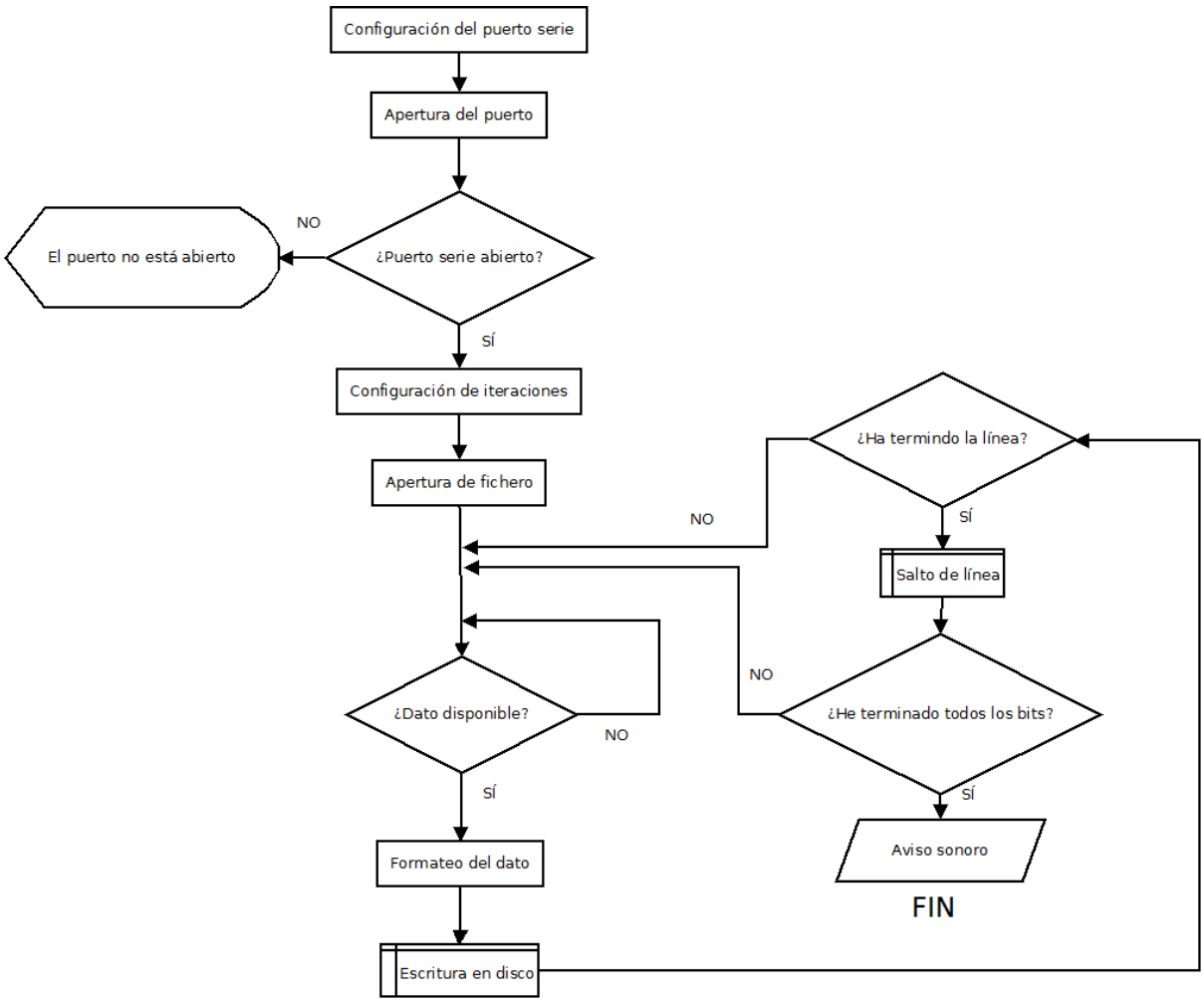


Ilustración 30 - Diagrama de flujo del programa en Python

Para ejecutar este programa solo es necesario copiar el código fuente incluido en el anexo y tener un intérprete de Python instalado en el sistema. Este intérprete es multiplataforma y es software libre, por lo que se puede descargar desde la web: http://www.python.org/

3.1.10. Análisis de los datos

Como ya se ha explicado anteriormente, es complicado o casi imposible saber si un número es o no aleatorio. Esto se debe a que solo podemos saber si un número es aleatorio si ha sido generado por un proceso aleatorio, es decir, en función de la fuente que lo ha generado, pero no observando el número en sí.

También se ha comentado previamente que, aunque no podamos saber si un número es aleatorio o no por sí mismo, sí que podemos estudiar secuencias lo suficientemente grandes desde el punto de vista de la estadística, ya que si tenemos un conjunto de números grande se puede esperar que cumplan ciertas reglas.

El propósito de los tests de aleatoriedad es comprobar si un conjunto de números cumple con ciertas condiciones desde el punto de vista estadístico y que no existan tendencias hacia uno de los dos valores posibles.

Cada test devuelve un valor de p que estará entre 0 y 1, pero no deberá ser ni muy cercano a 1 ni muy cercano a 0 si el conjunto de números es realmente aleatorio.

Según George Marsaglia, autor de los tests de DIEHARD [24]: "Los valores de p se obtienen mediante p=F(X) donde F se asume que es la distribución de la muestra de la variable aleatoria X. Sin embargo esto asume que F es solo una aproximación asintótica para la cual el ajuste será peor en las colas. Así, no se debe sorprender si obtiene valores de p cercanos a 0 o a 1 de forma ocasional. Cuando una cadena realmente falla, obtendrá unos valores de p de 0 o 1. De todos modos, no piense, como un estadístico podría pensar, que un valor de p menor que 0.025 o un p mayor que 0.975 significa que el generador de números aleatorio ha "fallado el tests a un nivel de 0.05". Este tipo de valores de p se obtienen entre los muchos que DIEHARD produce, incluso con buenos RNGs".

3.1.10.1. Sistema de puntuación

Para el análisis de los datos se ha empleado la batería de tests DIEHARD. Esta batería de tests lo componen 16 tests independientes, que realizan diferentes pruebas.

Se ha considerado que una prueba ha obtenido un resultado positivo si obtiene un valor de p perteneciente al conjunto [0.05, 0.95]. Por tanto, un valor de p fuera de dicho intervalo se considerará como un resultado negativo, implicando que el conjunto de números no ha superado la prueba.

Un mismo conjunto de números puede pasar un test y sin embargo no pasar otro. Esto ha llevado a establecer un sistema de puntuación cualitativo con el fin de poder comparar el rendimiento de diferentes generadores, que consiste en asignar una puntuación a los generadores de acuerdo a los siguientes parámetros:

- 0 puntos: el test da un resultado negativo.
- 0.5 puntos: en el caso de que un test dé varios valores "p", y algún valor sea incorrecto mientras que otros son correctos.
- 1 punto: en el caso de que el valor obtenido sea el adecuado.

De esta forma, la máxima puntuación que puede conseguir un conjunto de números aleatorios es 16 puntos. Representando estas puntuaciones podemos hacernos una idea del rendimiento de los diferentes generadores y de cómo afecta un parámetro específico a dicho rendimiento.

3.1.10.2. Descripción de los tests realizados

La batería de pruebas de aleatoriedad DIEHARD fue desarrollada por George Marsaglia en 1997 [24] y consta de 12 pruebas diferentes. A continuación se describen brevemente, ya que pueden ayudar a comprender tanto el concepto de

aleatoriedad como el de por qué, un mismo conjunto de números puede pasar unas pruebas y fallar otras.

1. Espaciado de cumpleaños: se elige un conjunto de puntos aleatorios en un gran intervalo. Los espacios entre los puntos deben seguir una distribución exponencial asintótica.
2. Permutaciones solapadas: analiza secuencias de cinco números aleatorios consecutivos. Los 120 ordenamientos posibles deben tener la misma probabilidad estadística de suceder.
3. Rangos de matrices: se elige algunos números de bits aleatorios para formar una matriz sobre {0, 1}. Después se determina el rango de la matriz y se cuentan los valores de los rangos obtenidos.
4. Prueba de los monos: Las secuencias de bits se interpretan como palabras. Se cuentan las palabras solapadas en una cadena y el número de palabras que no aparecen deberían ajustarse a alguna distribución conocida.
5. Contar los unos: consiste en contar los bits con valor 1 en cada conjunto elegido. El resultado se convierte en una palabra y se cuenta el número de veces que se repite esa palabra.
6. Prueba del aparcamiento: Se colocan círculos de valor unidad de forma aleatoria en un cuadrado de 100 por 100. Si el círculo se solapa con uno existente se vuelve a intentar. Después de 12000 intentos el número de círculos "aparcados" deberá seguir una distribución de tipo normal.
7. Prueba de la distancia mínima: Se colocan de forma aleatoria 8000 puntos en un cuadrado de 10000 por 10000. Después se encuentra la distancia mínima entre las parejas. El cuadrado de la distancia debe seguir una distribución exponencial.
8. Prueba de las esferas aleatorias: se eligen de forma aleatoria 4000 puntos en un cubo de 1000 de arista. Se centra una esfera en cada

punto cuyo radio es la distancia mínima al punto más cercano. El volumen de la esfera más pequeña debe ser un valor que siga una distribución exponencial con cierta media.

9. Prueba del extrusor: se multiplica 2^{31} por valores aleatorios de coma flotante en el intervalo [0, 1) hasta que se alcance 1. Se repite este proceso 100.000 veces. El número de valores de coma flotante necesarios para llegar a 1 debe seguir una distribución conocida.
10. El tests de los solapamientos: se genera una secuencia de valores de coma flotante en el rango [0, 1). Se añaden secuencias de 100 valores de coma flotante consecutivos. Las sumas deben seguir una distribución conocida con una sigma y media determinadas
11. Runs tests: se genera una secuencia larga de números aleatorios en el rango [0, 1). Se cuenta de forma ascendente y descendente la cantidad de secuencias invariantes del mismo bit.
12. El test de craps: se juegan una serie de juegos de craps (juego de dados) y se cuenta el número de victorias y el número de tiradas. Cada cuenta debe seguir una distribución determinada.

3.1.10.3. Pruebas previas de la batería DIEHARD

Antes de utilizar las pruebas de DIEHARD, se va a hacer un breve estudio de los resultados obtenidos con generadores de tipo PRNG que incluye el propio software, con el fin de comprobar su correcto funcionamiento.

Por ejemplo, mediante el generador determinista KISS, podemos generar un conjunto de números aleatorios. Para ello es necesario proporcionar 4 números enteros que se usarán como semilla para obtener la secuencia. Las semillas proporcionadas han sido: 2839405, 89467213, 76999, 584631

El fichero resultante será de tipo binario y se podrá visualizar y modificar con un editor hexadecimal.

```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff
00000000: 1b4e 0900 1310 5a24 8d10 f2ec fbc3 ab91
00000010: 032f d3ca 4c0b 9d14 0cb6 484b 5bd6 66cd
00000020: c3c9 5d2a fc7b 1582 459c c0ec d205 5d44
00000030: 7264 675c 7dfd 1bdf 2123 2632 437a 6b53
00000040: 98fb 7fa7 70db 079d a0e8 b6bd d08c 02a1
00000050: f107 9dd5 7176 9b3f 1dc6 c158 7e7e 209a
00000060: 801a fa5a 85eb ad80 e017 0610 0db9 7968
00000070: 10e0 c1a7 a3fa b614 af4d 4278 5f85 317c
00000080: 997e 6dad 8dc0 7f13 7a6d 496d 0503 ff78
00000090: c936 0a12 3250 b5ed 5abe 01bc a949 2311
000000a0: 3eb1 a21e 4fc9 76cf 3370 3441 da5e b743
000000b0: 8cd4 27cf 2d67 bb1b 228d bbc7 50a5 95bf
000000c0: f67b 9f56 1d3f 4db9 881a d539 a4b0 001f
000000d0: 767f c5ba f47b 0850 0d35 d421 6f20 ce80
000000e0: ca19 5bd5 83e8 563d 4f61 7bc7 bdaa c992
000000f0: b7a7 4b4a 79e0 f8ef e23f 64ca fe3a 5c8d
00000100: 50dc 6440 9a16 2b22 10c9 b734 2a34 eb04
00000110: c363 bde9 975e 9a34 9dd7 dfa5 80a8 5545
```

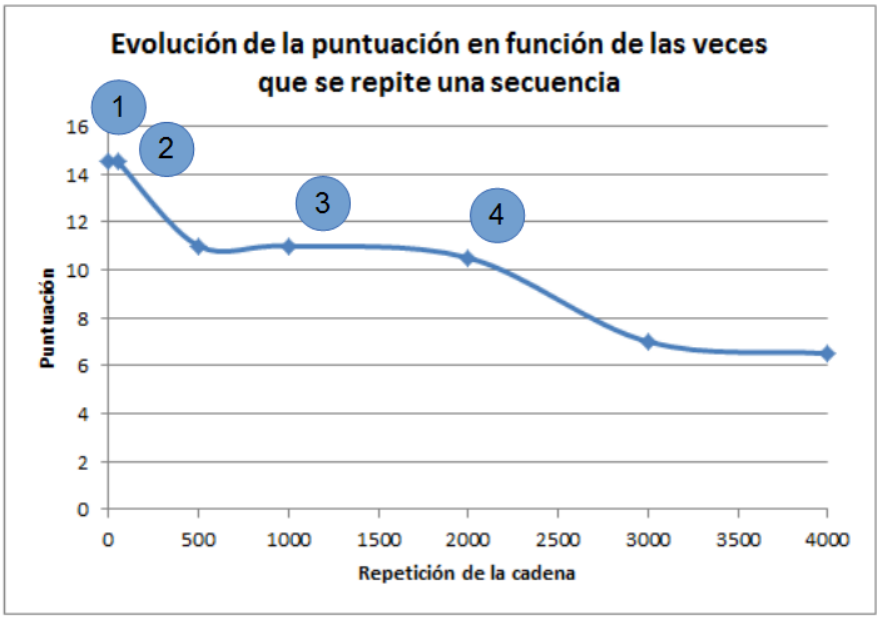
Ilustración 31 – Generación de un fichero binario mediante un PRNG (KISS)

```
87654321  0011 2233 4455 6677 8899 aabb ccdd eeff
00000000: cafe cafe cafe cafe cafe cafe cafe cafe
00000010: cafe cafe cafe cafe cafe cafe cafe cafe
00000020: cafe cafe cafe cafe cafe cafe cafe cafe
00000030: cafe cafe cafe cafe cafe cafe cafe cafe
00000040: cafe cafe cafe cafe cafe cafe cafe cafe
00000050: cafe cafe cafe cafe cafe cafe cafe cafe
00000060: cafe cafe cafe cafe cafe cafe cafe cafe
00000070: 10e0 c1a7 a3fa b614 af4d 4278 5f85 317c
00000080: 997e 6dad 8dc0 7f13 7a6d 496d 0503 ff78
00000090: c936 0a12 3250 b5ed 5abe 01bc a949 2311
000000a0: 3eb1 a21e 4fc9 76cf 3370 3441 da5e b743
000000b0: 8cd4 27cf 2d67 bb1b 228d bbc7 50a5 95bf
000000c0: f67b 9f56 1d3f 4db9 881a d539 a4b0 001f
000000d0: 767f c5ba f47b 0850 0d35 d421 6f20 ce80
000000e0: ca19 5bd5 83e8 563d 4f61 7bc7 bdaa c992
000000f0: b7a7 4b4a 79e0 f8ef e23f 64ca fe3a 5c8d
00000100: 50dc 6440 9a16 2b22 10c9 b734 2a34 eb04
00000110: c363 bde9 975e 9a34 9dd7 dfa5 80a8 5545
```

Ilustración 32 - Primera modificación del fichero

Si observamos la evolución de la puntuación en función de la repetición del número hexadecimal “CAFE” en el fichero, se puede apreciar como la puntuación desciende rápidamente con unas 500 repeticiones de la cadena pero se mantiene estable en esa puntuación hasta las 200 repeticiones. Esto vuelve a suceder más adelante con 3000 y 4000 repeticiones.

En la siguiente ilustración se puede apreciar cómo, además, las parejas 1-2 y 3-4 pasan unos tests similares. Este efecto puede deberse a que determinadas pruebas no son sensibles a ese tipo de repetición hasta un umbral determinado.



1 Original 1

		Resultado
1	BIRTHDAY SPACINGS TEST	1
2	THE OVERLAPPING 5-PERMUTATION TEST	0.5
3	BINARY RANK TEST for 31x31 matrices	1
4	BINARY RANK TEST for 32x32 matrices	1
5	BINARY RANK TEST for 6x8 matrices	1
6	THE BITSTREAM TEST	1
7	The tests OPSO, OQSO and DNA	1
8	COUNT-THE-1's TEST on a stream of bytes	1
9	COUNT-THE-1's TEST for specific bytes	1
10	PARKING LOT TEST	1
11	THE MINIMUM DISTANCE TEST	1
12	THE 3DSPHERES TEST	1
13	SQEEZE test	1
14	OVERLAPPING SUMS test	1
15	RUNS test	0.5
16	CRAPS TEST	0.5
	Número total de tests pasados	14.5

2 modificación 50 cafes

		Resultado
1	BIRTHDAY SPACINGS TEST	1
2	THE OVERLAPPING 5-PERMUTATION TEST	0.5
3	BINARY RANK TEST for 31x31 matrices	1
4	BINARY RANK TEST for 32x32 matrices	1
5	BINARY RANK TEST for 6x8 matrices	1
6	THE BITSTREAM TEST	1
7	The tests OPSO, OQSO and DNA	1
8	COUNT-THE-1's TEST on a stream of bytes	1
9	COUNT-THE-1's TEST for specific bytes	1
10	PARKING LOT TEST	1
11	THE MINIMUM DISTANCE TEST	1
12	THE 3DSPHERES TEST	1
13	SQEEZE test	1
14	OVERLAPPING SUMS test	1
15	RUNS test	0.5
16	CRAPS TEST	0.5
	Número total de tests pasados	14.5

3 modificado 1000 cafes

		Resultado
1	BIRTHDAY SPACINGS TEST	1
2	THE OVERLAPPING 5-PERMUTATION TEST	0.5
3	BINARY RANK TEST for 31x31 matrices	1
4	BINARY RANK TEST for 32x32 matrices	0
5	BINARY RANK TEST for 6x8 matrices	0
6	THE BITSTREAM TEST	1
7	The tests OPSO, OQSO and DNA	1
8	COUNT-THE-1's TEST on a stream of bytes	0.5
9	COUNT-THE-1's TEST for specific bytes	0
10	PARKING LOT TEST	1
11	THE MINIMUM DISTANCE TEST	1
12	THE 3DSPHERES TEST	1
13	SQEEZE test	1
14	OVERLAPPING SUMS test	1
15	RUNS test	0
16	CRAPS TEST	1
	Número total de tests pasados	11

4 modificado 2000 cafes

		Resultado
1	BIRTHDAY SPACINGS TEST	1
2	THE OVERLAPPING 5-PERMUTATION TEST	0
3	BINARY RANK TEST for 31x31 matrices	1
4	BINARY RANK TEST for 32x32 matrices	0
5	BINARY RANK TEST for 6x8 matrices	0
6	THE BITSTREAM TEST	1
7	The tests OPSO, OQSO and DNA	1
8	COUNT-THE-1's TEST on a stream of bytes	0.5
9	COUNT-THE-1's TEST for specific bytes	0
10	PARKING LOT TEST	1
11	THE MINIMUM DISTANCE TEST	1
12	THE 3DSPHERES TEST	1
13	SQEEZE test	1
14	OVERLAPPING SUMS test	1
15	RUNS test	0
16	CRAPS TEST	1
	Número total de tests pasados	10.5

Ilustración 33 - Evolución de la puntuación de los tests en función de la cantidad de repeticiones de un número en el fichero

3.2. Pruebas y resultados

Las pruebas que aquí se plantean tienen dos propósitos. El primero de ellos es el de estudiar cómo varía la respuesta del circuito a diferentes cambios en su implementación o en su diseño.

Debido al carácter asíncrono del diseño se pueden apreciar cambios muy significativos en las señales críticas no solo por cambios en su diseño, sino también por cambios en factores como el dónde se implementan o la temperatura ambiente. Esto significa que, un mismo circuito tendrá un comportamiento completamente diferente si se instancia en una parte de la FPGA o en otra.

El segundo de los propósitos es el de poder usar estos resultados para poder mejorar el rendimiento del TRNG diseñado. Para ello se irán seleccionando los mejores resultados, esperando así que eso se traduzca en un TRNG de alta calidad.

3.2.1. Consideraciones previas sobre los estudios y sus resultados

Muchos de los estudios se han realizado observando el comportamiento de la frecuencia de S0_s y no el resultado de los tests. Eso se debe a que el resultado de los tests solo proporciona información sobre la calidad de los 100 millones de bits que haya generado el TRNG en ese estudio.

Aunque se ha visto que el resultado de los tests tiene una alta repetitividad, esta puntuación es un factor que nos proporciona poca información sobre qué es lo que está pasando realmente dentro de la FPGA y en muchas ocasiones se vuelve compleja y difícil de relacionar con otros parámetros.

Una alternativa a la de comparar a los generadores en función de la puntuación obtenida, es la de observar la variación de los osciladores, midiendo tanto su frecuencia como el intervalo de incertidumbre de la misma. Esta opción es inviable ya que en muchas ocasiones los osciladores superan los 100 MHz y los osciloscopios disponibles están limitados a esa misma frecuencia. Esto hace que las

mediciones a esas frecuencias límites no sean fiables y no se deban tomar como resultados válidos.

A pesar de que sea posible medir los dos osciladores sí que resulta posible medir la diferencia entre las frecuencias de las señales, proporcionada por S0_s. Por ello se ha elegido la opción de estudiar la frecuencia de la dicha señal.

Los cambios de cada estudio se detallan en cada uno. Como estrategia general se han modificado las posiciones de los bloques críticos. Estos son: ambos osciladores con todas sus puertas y las señales S0_s y C0_s.

3.2.2. Repetitividad de los resultados

Debido a la gran cantidad de factores que afectan al TRNG se podría pensar que los datos obtenidos son difíciles de volver a obtener. No obstante, las pruebas que se presentan a continuación tienen una alta repetitividad.

En el caso particular de las señales S0_s y la de los osciladores los resultados obtenidos son siempre iguales en las mismas condiciones de trabajo. No obstante, es importante recalcar que en muchas ocasiones S0_s ha tenido una frecuencia cambiante, y se ha decidido apuntar un resultado intermedio entre la frecuencia máxima y la mínima observadas.

En el caso de la puntuación de los tests la repetitividad no es total. Eso es algo normal incluso con los propios generadores deterministas ya que las pruebas se realizan sobre un conjunto de números finito. Este conjunto de números puede que falle algún tests incluso siendo un buen generador. George Marsaglia [24], autor de la batería de tests DIEHARD advierte que esto es normal que suceda y que puede suceder que otro conjunto de números, producido por el mismo generador pase la prueba.

No obstante este no suele ser el caso y los generadores suelen tener una puntuación que no varía del 10% en el peor de los casos

3.2.3. Relación entre la frecuencia de S0_s y los resultados de los tests

En las pruebas que se van a realizar a continuación se ha analizado la frecuencia de S0_s. Esto se debe a que esta señal guarda una relación muy fuerte con la aleatoriedad del sistema.

Como el funcionamiento del generador está basado en el muestreo del jitter de los osciladores, las señales de estos osciladores deben ser muy parecidas, y por tanto, el valor de S0_s debe ser bajo.

En la siguiente gráfica se muestra cómo los osciladores que tienen una S0_s con una frecuencia baja tienen un mejor comportamiento que los generadores cuya señal S0_s tiene una alta frecuencia

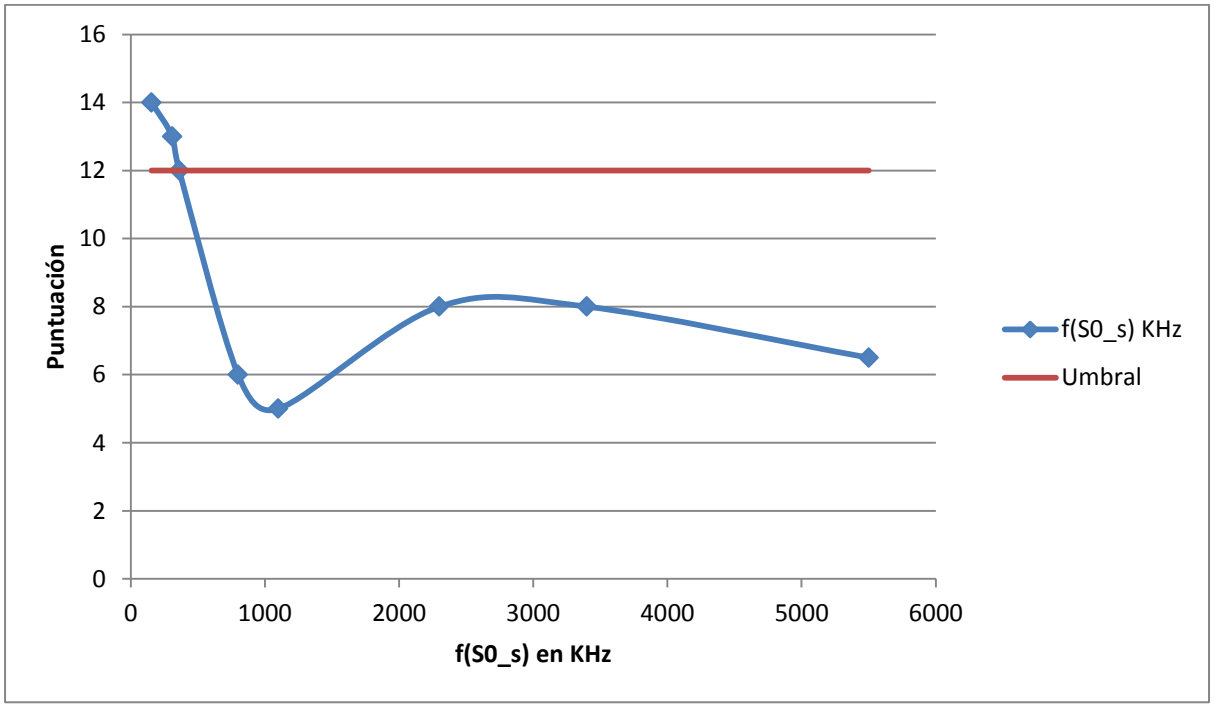


Ilustración 34 - Relación entre f(S0_s) y la puntuación obtenida

Debido a esta relación entre la frecuencia de S0_s y la puntuación obtenida en los tests se va a proceder a analizar dicha señal en las pruebas posteriores.

3.2.4. Cambio en la complejidad de los osciladores.

En esta prueba se ha modificado la cantidad de puertas NOT de las que están compuestos los osciladores de forma sucesiva, dejando en manos del sintetizador la colocación de las mismas.

3.2.4.1. Variación de la frecuencia de los osciladores

En el siguiente gráfico se muestra la evolución de la frecuencia de clk0_s en función del número de puertas NOT. Como ya se había adelantado, la frecuencia de los osciladores disminuye a medida que aumenta el número de puertas lógicas de las que están compuestas. Los valores de las frecuencias para 1, 3 y 5 puertas no se encuentran reflejados en la gráfica, ya que el osciloscopio usado está limitado a una velocidad de 100 MHz.

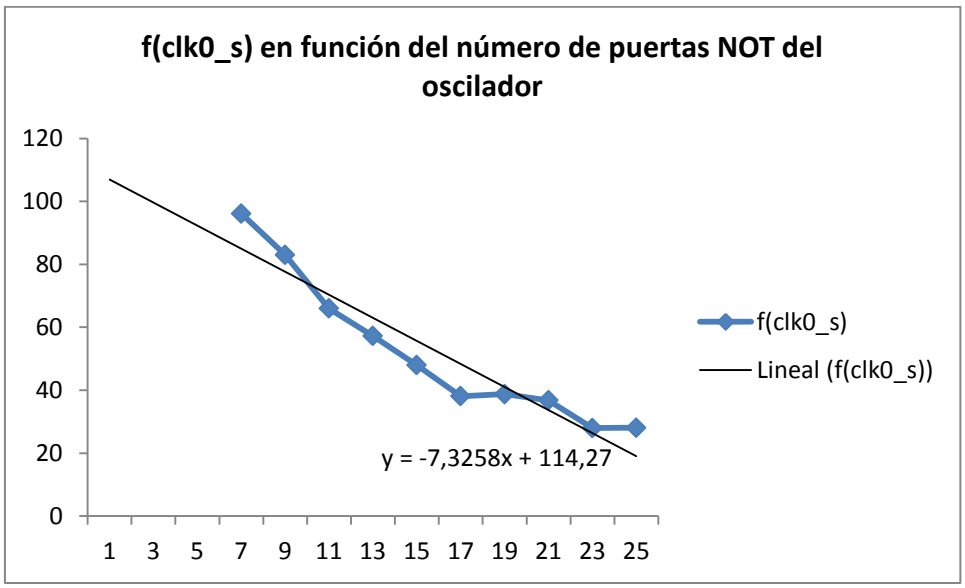


Ilustración 35 - f(clk0_s) en función del número de puertas NOT del oscilador

Como cada puerta tiene asociado un rutado, y que tanto cada puerta, como cada rutado añaden un valor fijo en el retardo de la oscilación, podemos suponer que el cambio de la frecuencia seguirá una tendencia lineal.

De esta forma, podemos hacer una estimación de cuánto retardo añaden una puerta NOT y una puerta AND.

Puesto que la regresión lineal nos da un resultado de:

$$y = -7.3258x + 114.27$$

Por lo que el retardo de una puerta AND más el rutado necesario será:

$$Si\ x = 0 \rightarrow y = 114.27\ MHz,$$

$$Retardo_{AND} = \frac{1}{114.27} = 8.75\ ns$$

Y el retardo de una puerta NOT junto a su rutado necesario será, aproximadamente:

$$Retardo_{NOT} = \frac{1}{7.325} = 137\ ns$$

3.2.4.2. *Variación de la frecuencia de S0_s*

Además del cambio en la frecuencia de los osciladores es posible estudiar el cambio en S0_s.

Como tendencia general, se puede ver cómo a medida que desciende el valor de la frecuencia de los osciladores, se obtiene una señal mucho más estable entre los mismos, ya que el valor de la frecuencia de S0_s desciende:

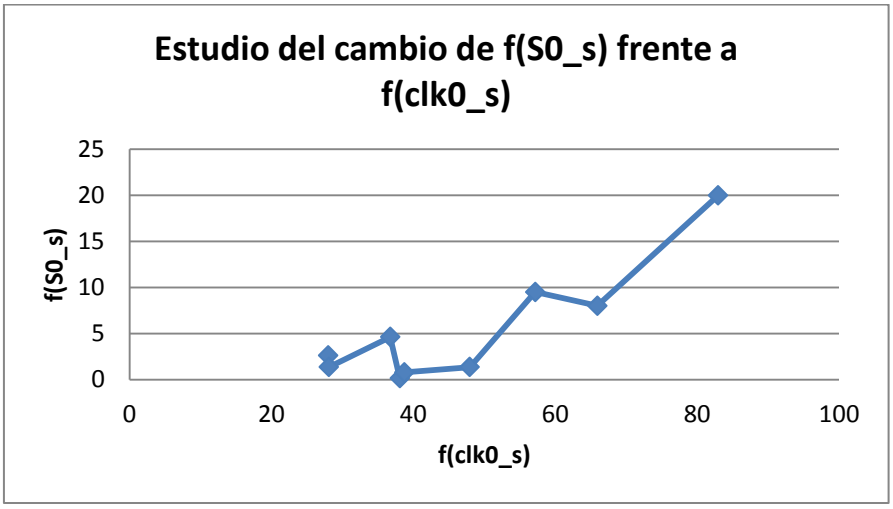


Ilustración 36 - Estudio de f (S0_o) frente a f(clk0_s)

3.2.5. Cambio en la distribución de los osciladores

En esta prueba se ha comprobado el cambio en la señal S0_s al modificar su distribución en la instanciación. Es importante recalcar que no se ha modificado el diseño de los bloques críticos. Esto significa que no se ha modificado ni la cantidad de puertas de los osciladores ni su diseño. Los cambios que se producen en la señal S0_s se deben solo al cambio en la disposición relativa de los bloques.

En todos los casos se ha mantenido la posición absoluta de los bloques (el origen se sitúa en X0Y0) y se ha modificado la posición relativa entre sí. Para entender el sistema de coordenadas usado en la FPGA Spartan-3E, se proporciona la siguiente imagen, obtenida de [25]:

Ilustración 37 - Numeración de los "Slices" en diferentes modelos de FPGA, entre ellos la Spartan-3E, obtenida de [25]

Distribución	frecuencia S0_s
1	170 KHz
2	400 KHz
3	600 kHz
4	1500kHz
5	1500 KHz
6	2700 KHz

Tabla 5 - Cambio en la frecuencia de S0_s según diferentes distribuciones de los bloques críticos

Como podemos ver, hay una gran diferencia dependiendo de la distribución de los osciladores aun siendo el mismo circuito desde el punto de vista lógico.

A continuación se especifican las diferentes distribuciones usadas en esta experiencia:

Distribución 1	Distribución 2	Distribución 3
INST "clk0/a_s1" RLOC = X0Y0; INST "clk0/b_s1" RLOC = X0Y1; INST "clk0/c_s1" RLOC = X1Y0; INST "clk0/d_s1" RLOC = X1Y1; INST "clk0/e_s1" RLOC = X0Y2; INST "clk0/f_s1" RLOC = X0Y3; INST "clk1/a_s1" RLOC = X4Y0; INST "clk1/b_s1" RLOC = X4Y1; INST "clk1/c_s1" RLOC = X5Y0; INST "clk1/d_s1" RLOC = X5Y1; INST "clk1/e_s1" RLOC = X4Y2; INST "clk1/f_s1" RLOC = X4Y3; INST "sampler/S0_s" RLOC = X3Y3; INST "sampler/C0_s" RLOC = X3Y2; INST "clk0/a_s1" RLOC_ORIGIN = X0Y0;	INST "clk0/a_s1" RLOC = X0Y0; INST "clk0/b_s1" RLOC = X0Y1; INST "clk0/c_s1" RLOC = X0Y2; INST "clk0/d_s1" RLOC = X0Y3; INST "clk0/e_s1" RLOC = X0Y4; INST "clk0/f_s1" RLOC = X0Y5; INST "clk1/a_s1" RLOC = X2Y0; INST "clk1/b_s1" RLOC = X2Y1; INST "clk1/c_s1" RLOC = X2Y2; INST "clk1/d_s1" RLOC = X2Y3; INST "clk1/e_s1" RLOC = X2Y4; INST "clk1/f_s1" RLOC = X2Y5; INST "sampler/S0_s" RLOC = X1Y0; INST "sampler/C0_s" RLOC = X1Y1; INST "clk0/a_s1" RLOC_ORIGIN = X0Y0;	INST "clk0/a_s1" RLOC = X0Y0; INST "clk0/b_s1" RLOC = X1Y0; INST "clk0/c_s1" RLOC = X0Y2; INST "clk0/d_s1" RLOC = X1Y2; INST "clk0/e_s1" RLOC = X0Y4; INST "clk0/f_s1" RLOC = X1Y4; INST "clk1/a_s1" RLOC = X0Y1; INST "clk1/b_s1" RLOC = X1Y1; INST "clk1/c_s1" RLOC = X0Y3; INST "clk1/d_s1" RLOC = X1Y3; INST "clk1/e_s1" RLOC = X0Y5; INST "clk1/f_s1" RLOC = X1Y5; INST "sampler/S0_s" RLOC = X2Y3; INST "sampler/C0_s" RLOC = X2Y2; INST "clk0/a_s1" RLOC_ORIGIN = X0Y0;
Distribución 4	**Distribución 5**	**Distribución 6**
INST "clk0/a_s1" RLOC = X0Y0; INST "clk0/b_s1" RLOC = X0Y1; INST "clk0/c_s1" RLOC = X1Y0; INST "clk0/d_s1" RLOC = X1Y1; INST "clk0/e_s1" RLOC = X0Y2; INST "clk0/f_s1" RLOC = X0Y3; INST "clk1/a_s1" RLOC = X0Y4; INST "clk1/b_s1" RLOC = X0Y5; INST "clk1/c_s1" RLOC = X1Y4; INST "clk1/d_s1" RLOC = X1Y5; INST "clk1/e_s1" RLOC = X0Y6; INST "clk1/f_s1" RLOC = X0Y7; INST "sampler/S0_s" RLOC = X3Y3; INST "sampler/C0_s" RLOC = X3Y2;	INST "clk0/a_s1" RLOC = X0Y0; INST "clk0/b_s1" RLOC = X0Y1; INST "clk0/c_s1" RLOC = X1Y0; INST "clk0/d_s1" RLOC = X1Y1; INST "clk0/e_s1" RLOC = X0Y2; INST "clk0/f_s1" RLOC = X0Y3; INST "clk1/a_s1" RLOC = X0Y4; INST "clk1/b_s1" RLOC = X0Y5; INST "clk1/c_s1" RLOC = X1Y4; INST "clk1/d_s1" RLOC = X1Y5; INST "clk1/e_s1" RLOC = X0Y6; INST "clk1/f_s1" RLOC = X0Y7; INST "sampler/S0_s" RLOC = X1Y3; INST "sampler/C0_s" RLOC = X1Y2;	INST "clk0/a_s1" RLOC = X0Y0; INST "clk0/b_s1" RLOC = X0Y1; INST "clk0/c_s1" RLOC = X1Y0; INST "clk0/d_s1" RLOC = X1Y1; INST "clk0/e_s1" RLOC = X0Y2; INST "clk0/f_s1" RLOC = X0Y3; INST "clk1/a_s1" RLOC = X2Y0; INST "clk1/b_s1" RLOC = X2Y1; INST "clk1/c_s1" RLOC = X3Y0; INST "clk1/d_s1" RLOC = X3Y1; INST "clk1/e_s1" RLOC = X2Y2; INST "clk1/f_s1" RLOC = X2Y3; INST "sampler/S0_s" RLOC = X3Y3; INST "sampler/C0_s" RLOC = X3Y2;

3.2.6. Cambio del lugar de instanciación de S0_s

En esta prueba se han mantenido los osciladores instanciados en la misma posición y manteniendo su orden (Véase distribución número 1 en el estudio sobre la distribución).

No se ha apreciado cambios relevantes al modificar el punto de instanciación de S0_s. Ya que no parece que sea un factor determinante, S0_s se instanciará en X3Y3 a partir de ahora.

S0				
X	Y	distancia	f	
3	3	4.242641	180	KHz
14	14	19.79899	170	KHz
24	24	33.94113	180	KHz
34	34	48.08326	180	KHz
44	44	62.2254	170	KHz
54	54	76.36753	160	KHz

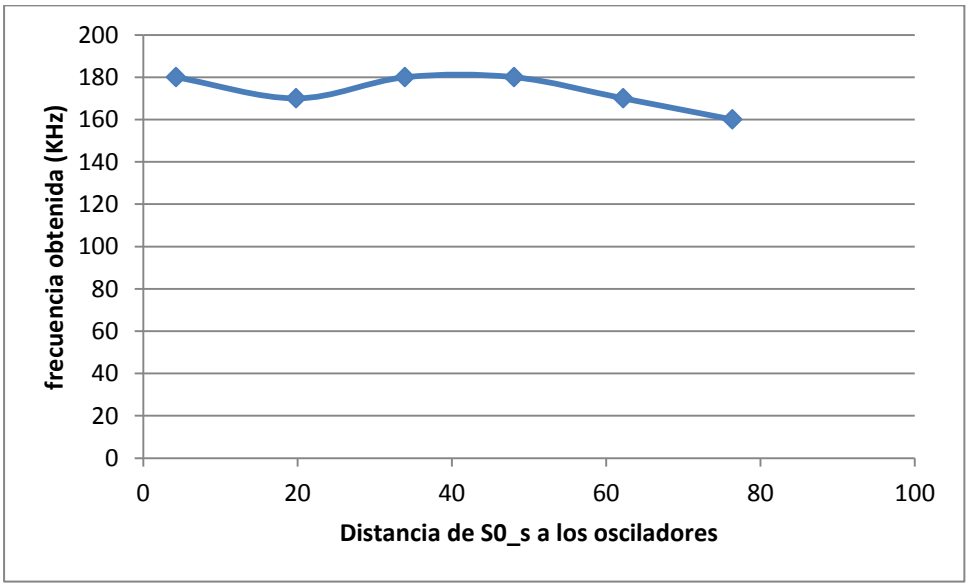
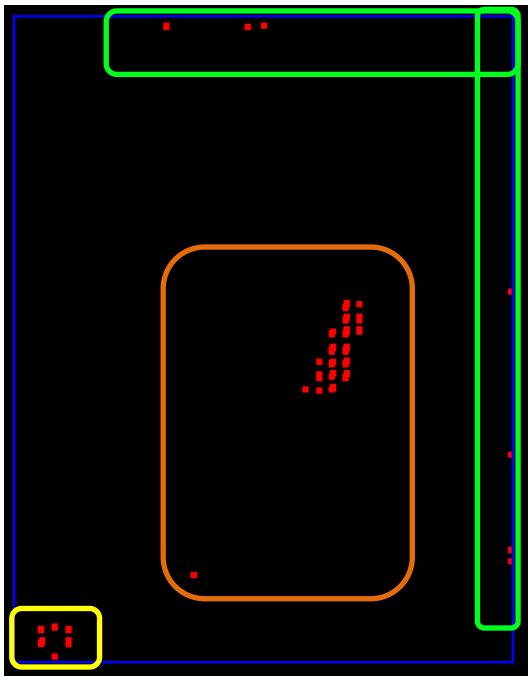


Ilustración 38 - Cambio en la frecuencia de S0_s en función del lugar de su instanciación

3.2.7. Cambio de otras partes del circuito

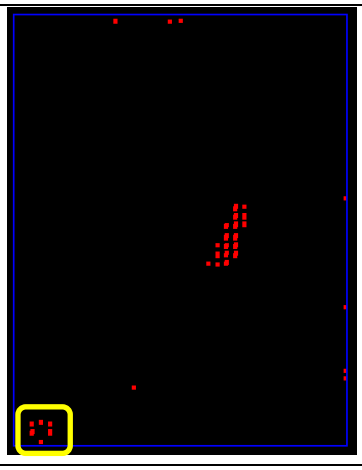
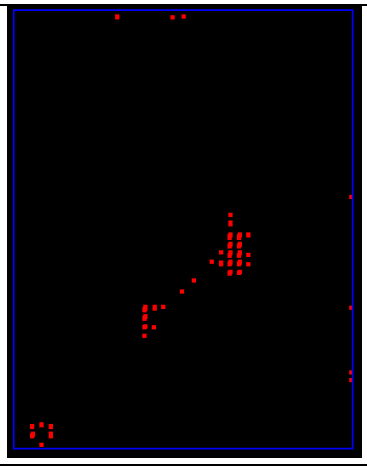
En este caso se estudia el cambio en la frecuencia de S0_s en función de alteraciones en el resto del circuito, manteniendo las partes críticas constantes.



- Verde: Salidas y entradas de la FPGA
- Naranja: circuitos no críticos
- Amarillo: Circuitos críticos

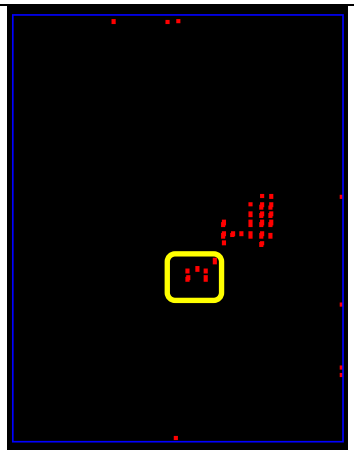
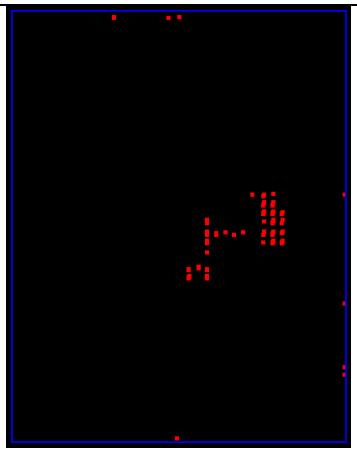
Modelo A: Preset = "00000000"

Modelo B: Preset = "11111111"

Modelo A	Modelo B
	
132 KHz	125 KHz

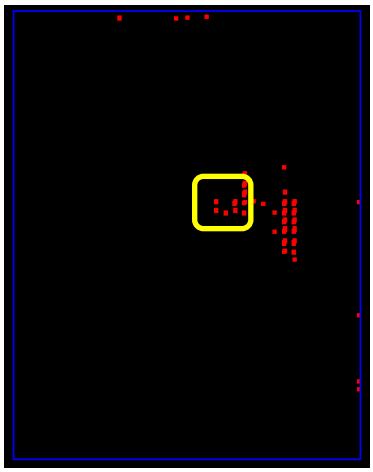
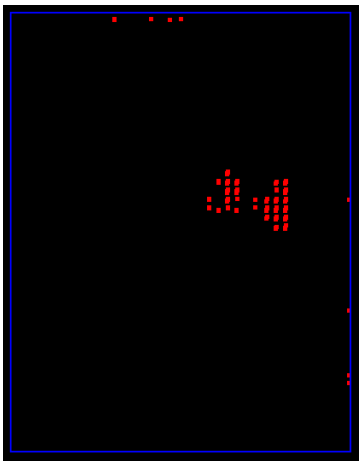
Porcentaje de cambio: 5.6%

Origen de la RPM: X34 Y34

Modelo A	Modelo B
	
125KHz	80 KHz

Porcentaje de cambio: 56.2%

Origen de la RPM: X39 Y39

Modelo A	Modelo B
	
600 KHz	4000KHz

Porcentaje de cambio: 566%

Como se puede apreciar, a medida que los osciladores se acercan más al resto de los circuitos, éstos se ven más afectados por cambios en los circuitos que los rodean. Esto permite sacar la conclusión de que además de los factores ya dichos, los osciladores se verán afectados por los circuitos que les rodean.

3.2.8. Cambio en la separación de los osciladores

Esta prueba ha consistido en comprobar cómo varía la frecuencia S0_s en función del número de SLICES de separación entre cada uno de los osciladores, es decir, el número de bloques programables de la FPGA que hay entre cada uno de ellos.

En este caso se ha mantenido el diseño y la distribución de los osciladores (distribución número 1) y se han ido alejando, de forma progresiva los osciladores entre sí.

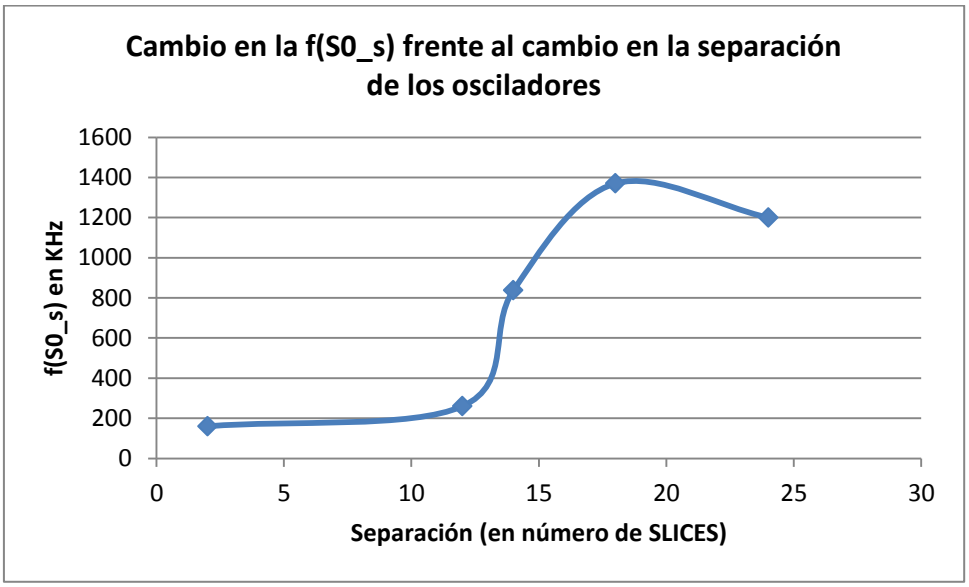


Ilustración 39 - Cambio en la f (S0_s) frente al cambio en la separación de los osciladores

Como se puede apreciar en el gráfico, la frecuencia cambia de forma repentina al distanciarse mucho un oscilador del otro, esto es debido a que la señal tendrá mucho más recorrido para uno de los dos y esto se traduce en una diferencia muy grande en la frecuencia de ambos.

3.2.9. Cambio del lugar de la instanciación de los osciladores

En esta prueba se ha optado por, manteniendo el diseño estudiado hasta ahora y respetando las distancias relativas entre los componentes, instanciar el bloque de componentes críticos en diferentes partes de la FPGA.

Es importante señalar que aunque se muestra una gráfica continua, existen muchos bloques en los que la instanciación de este diseño no es viable, bien porque no existe el bloque o porque no es posible el rutado de los mismos.

De todas formas se ha conseguido obtener unos 30 puntos repartidos de forma regular por gran parte de la superficie de la FPGA.

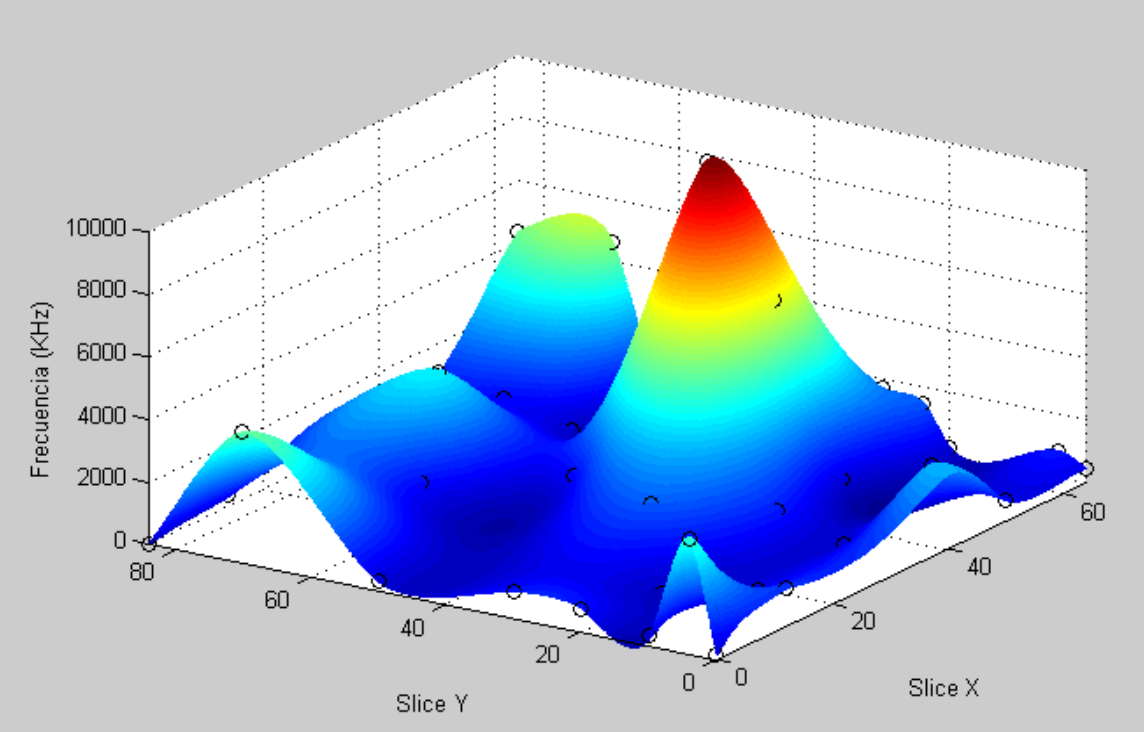


Ilustración 40 - Cambio en f (S0_s) en función de la posición de la FPGA

3.2.10. Temperatura

Se han realizado distintas pruebas de los cambios que efectúa la temperatura en las señales críticas del generador. Para ello se ha introducido la FPGA debidamente protegida en un congelador a -18ºC y se han tomado medidas cada dos minutos.

A continuación se presentan los dos análisis que se han efectuado sobre la influencia de la temperatura: la influencia en los osciladores y la influencia en S0_s.

3.2.10.1. Variación de la frecuencia de los osciladores

Para realizar el análisis de los osciladores ha sido necesario disminuir la frecuencia de los mismos hasta una frecuencia medible por las herramientas disponibles, es decir, 100 MHz. Se ha aumentado el número de puertas hasta las 11 puertas NOT, obteniendo una señal con una frecuencia de 70 MHz.

En general no se han podido apreciar cambios relevantes ni en el período de la señal ni en su amplitud. Esto se debe, en gran medida, a que a tales frecuencias es difícil captar variaciones de varios KHz ya que representan un porcentaje muy pequeño de la frecuencia total medida. Debido a ello se ha optado por realizar el análisis de S0_s, que es una señal más lenta y por tanto será más fácil realizar mediciones del cambio en la frecuencia.

3.2.10.2. Variación de la frecuencia de la señal S0_s

A continuación se presenta la evolución de la frecuencia de la señal S0_s en función de la temperatura. Ya que la frecuencia es cambiante se ha optado por representar tanto la frecuencia mínima como la máxima vistas en un período de 10 segundos.

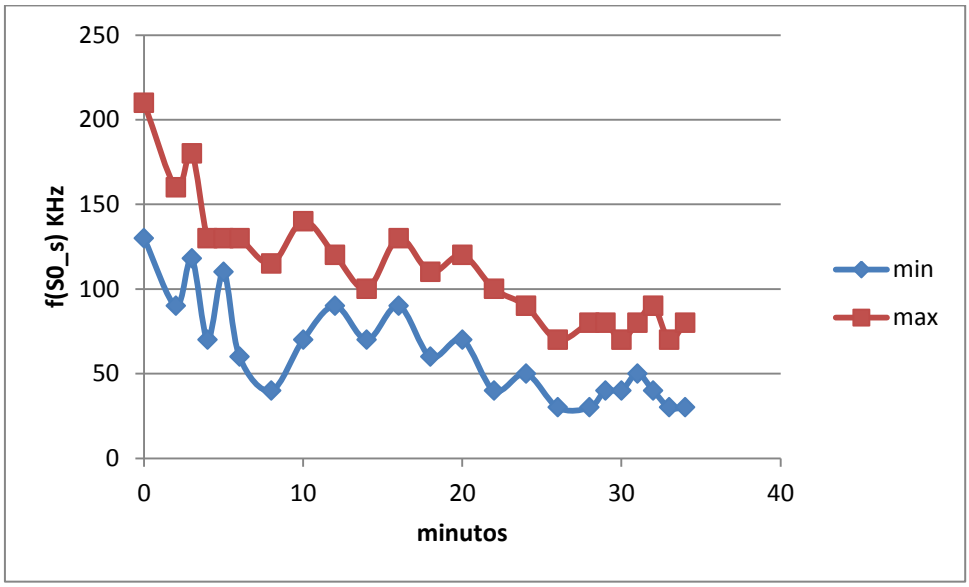


Ilustración 41 - Variación en f(S0_s) en función del tiempo que pasa la FPGA expuesta a -18ºC

Como se puede ver, la frecuencia de S0_s baja a medida que aumenta el tiempo expuesto a -18ºC. Esto se puede interpretar como que las señales de los osciladores serán más semejantes entre sí.

3.2.11. Resultados en otras FPGAs

En esta prueba se ha intentado poner de manifiesto si los cambios y los parámetros que se han visto previamente afectan de igual manera a otra FPGA del mismo modelo que la empleada en el resto de estudios.

Para ello se ha instanciado el mismo diseño en dos FPGAs del mismo modelo y se ha medido la frecuencia de la señal S0_s. Simplemente, comparando dos casos es posible ver cómo la instanciación en la primera FPGA da una baja frecuencia en S0_s, en la segunda genera una alta frecuencia.

	Distribución 1	**Distribución 2**
	FPGA Numero 1	**FPGA Numero 2**
Instanciación 1	170 KHz	1400 KHz
Instanciación 2	2700 KHz	140 KHZ

Estos resultados muestran como, en general, existe una muy baja capacidad de portabilidad de los diseños a otras FPGAs incluso del mismo modelo ya que aunque se mantenga constante el diseño y los lugares de instanciación, no se mantendrán constantes las frecuencias de los osciladores.

Esto se puede explicar por factores de fabricación: los transistores son un poco diferentes, las conductividades de las pistas no son exactamente iguales, etc. Y esto implica que sea realizar un análisis del diseño cada vez que se desee portar a otra FPGA incluso del mismo modelo.

3.2.12. Mejoras de los resultados mediante análisis posterior. Con Neumann.

En este caso se ha estudiado cómo influye la técnica de mejora de resultados sobre el TRNG, así como el cambio en el tiempo que se tardan en generar y transmitir 100 millones de bits.

A continuación se presentan dos ejemplos a modo ilustrativo de la mejora de los rendimientos en los generadores. Como se puede apreciar, la mejora de los resultados es sustancial, ya que el mismo generador pasa de puntuaciones de puntuaciones malas a puntuaciones muy buenas:

	Preset 00000001	Sin Von Neumann	Con Von Neumann
		Resultado	Resultado
1	BIRTHDAY SPACINGS TEST	1	1
2	THE OVERLAPPING 5-PERMUTATION TEST	1	1
3	BINARY RANK TEST for 31x31 matrices	1	1
4	BINARY RANK TEST for 32x32 matrices	1	1
5	BINARY RANK TEST for 6x8 matrices	0	1
6	THE BITSTREAM TEST	0	1
7	The tests OPSO, OQSO and DNA	0	1
8	COUNT-THE-1's TEST on a stream of bytes	0	1
9	COUNT-THE-1's TEST for specific bytes	0	1
10	PARKING LOT TEST	0	1
11	THE MINIMUM DISTANCE TEST	1	1
12	THE 3DSPHERES TEST	1	0
13	SQEEZE test	0	1
14	OVERLAPPING SUMS test	1	1
15	RUNS test	1	1
16	CRAPS TEST	0	1
	Número total de tests pasados	8	15

	Preset 00000010	Sin Von Neumann	Con Von Neumann
		Resultado	Resultado
1	BIRTHDAY SPACINGS TEST	1	1
2	THE OVERLAPPING 5-PERMUTATION TEST	1	1
3	BINARY RANK TEST for 31x31 matrices	1	1
4	BINARY RANK TEST for 32x32 matrices	0	1
5	BINARY RANK TEST for 6x8 matrices	0	1
6	THE BITSTREAM TEST	0	1
7	The tests OPSO, OQSO and DNA	0	1
8	COUNT-THE-1's TEST on a stream of bytes	0	0.5
9	COUNT-THE-1's TEST for specific bytes	0	1
10	PARKING LOT TEST	0	1
11	THE MINIMUM DISTANCE TEST	1	1
12	THE 3DSPHERES TEST	0	1
13	SQEEZE test	0	1
14	OVERLAPPING SUMS test	1	1
15	RUNS test	1	1
16	CRAPS TEST	0	0.5
	Número total de tests pasados	6	15

Tabla 6 - Cambio en la puntuación empleando la técnica de Von Neumann

No obstante, esta mejora en el rendimiento tiene como principal consecuencia la disminución de la tasa de transferencia de bits, ya que se ha de rechazar muchos de los bits generados.

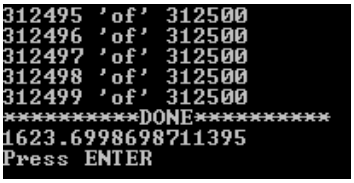
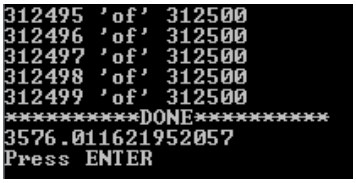
Sin Von Neumann	Con Von Neumann
	

Tabla 7 - Comparación de tiempos según se emplee o no la técnica de Von Neumann

El dato justo después de la palabra en inglés DONE indica el tiempo en segundos que ha tardado el sistema en generar y transmitir los 100 millones de bits. En el primer caso se ha tardado 27 minutos aproximadamente, mientras que en el caso de emplear Von Neumann son necesarios 59 minutos, por lo que la tasa de generación y transmisión se reduce a la mitad.

3.2.13. Resumen del rendimiento y comparación con generadores PRNG

En las pruebas presentadas se muestra cómo se modifica el comportamiento del TRNG en general y de los osciladores y señales críticas en particular. Una vez reunidas todas las pruebas, se puede concluir que este diseño se ve altamente influenciado por factores como su diseño (diferentes implementaciones de los osciladores), la distribución y el lugar de implementación. También afecta, aunque en menor medida, la temperatura a la que se encuentra la FPGA.

Todas estas interdependencias hacen que, aunque el diseño sea portable a otras FPGAs, se requiera tiempo en analizar las señales críticas y en probar diferentes distribuciones ya que el rendimiento del TRNG depende en gran medida de estas señales.

Los resultados obtenidos sin hacer uso de la técnica de mejora de Von Neumann han sido muy variables. Se han llegado a obtener muy buenos generadores que, tan solo cambiando el lugar de instanciación, han pasado a generar números con muy baja aleatoriedad.

No obstante, controlando las señales críticas y haciendo uso de Von Neumann, se llegan a obtener unos resultados muy buenos en la generación de números. En la siguiente tabla se muestra una comparativa del TRNG haciendo uso de Von Neumann y diferentes generadores deterministas:

		TRNG	Generadores deterministas			
		Von Neumann	KISS	Microsoft Fortran	MWC	MWC de 16 bits
1	BIRTHDAY SPACINGS TEST	1	1	1	1	1
2	THE OVERLAPPING 5-PERMUTATION TEST	1	0.5	1	1	1
3	BINARY RANK TEST for 31x31 matrices	1	1	1	1	1
4	BINARY RANK TEST for 32x32 matrices	1	1	1	1	1
5	BINARY RANK TEST for 6x8 matrices	1	1	1	1	1
6	THE BITSTREAM TEST	1	1	1	1	1
7	The tests OPSO, OQSO and DNA	1	1	1	1	1
8	COUNT-THE-1's TEST on a stream of bytes	1	1	1	1	1
9	COUNT-THE-1's TEST for specific bytes	1	1	1	1	1
10	PARKING LOT TEST	1	1	0	1	0
11	THE MINIMUM DISTANCE TEST	1	1	1	1	1
12	THE 3DSPHERES TEST	0	1	1	1	1
13	SQEEZE test	1	1	1	1	1
14	OVERLAPPING SUMS test	1	1	0	1	1
15	RUNS test	1	0.5	1	1	1
16	CRAPS TEST	1	0.5	1	1	1
	Número total de tests pasados	15	14.5	14	16	15

Ilustración 42 - Comparación del TRNG con otros generadores deterministas

4. Conclusiones

Una vez estudiados todos los parámetros que afectan al diseño e implementación del TRNG propuesto es necesario hacer un balance general de los resultados obtenidos.

En primer lugar, hay que recalcar que se ha conseguido cumplir con el principal objetivo del trabajo que es generar números aleatorios con el TRNG propuesto. Además, los resultados obtenidos tienen una calidad similar a los obtenidos mediante PRNGs, pudiendo decir que este TRNG se comporta realmente como tal.

No obstante, este diseño de generador tiene una serie inconvenientes que hacen que no sea óptimo. El mayor de estos problemas es la necesidad de realizar un análisis exhaustivo de las señales de la FPGA durante la fase de diseño e implementación.

Esto se debe a que los números generados se ven muy influenciados por la frecuencia de ciertas señales, denominadas críticas, que, además se verán afectadas por muchos parámetros. Esto parámetros son:

- Diseño de los osciladores
- Lugar de implementación y distribución de los osciladores
- Modificación de bloques cercanos a los osciladores
- La temperatura
- La FPGA en la que se implementa el TRNG

Todo ello hace que la portabilidad del TRNG sea baja, es decir, que se necesite una gran cantidad de trabajo para poder implementar el generador en otra FPGA.

Por lo tanto, se puede concluir que, a pesar de que este diseño de TRNG es viable, no es un sustituto óptimo a los diseños que usan PLLs.

5. Bibliografía

[1] P. J. Davis, "What is meant by the word Random" in Mathematics and common sense, 2006.

[2] C. J, Information, randomness and incompleteness, Teaneck, USA: World Scientific Publishing Co., 1987.

[3] B. Jun y P. Kocher, «The Intel random number generator, White paper for Intel Corporation, Cryptography Research Inc.,» 1999.

[4] A. Stripp, The Enigma Machine: Its Mechanism and Use, 1993.

[5] A. C. Yao, Theory and application of trapdoor functions, CHicago: IEEE, 1982.

[6] Shannon, «A Mathematical Theory of Communication,» vol. XXVII, nº 3, 1948.

[7] S. Ihara, Information Theory for Continuous Systems, Octubre: World Scientific Pub Co Inc, 1993.

[8] W. Killmann y W. Schindler, Functionality Classes and Evaluation Methodology for True (Physical) Random Number Generators, Bonn: Bundesamt für Sicherheit in der Informationstechnik (BSI), 2001.

[9] W. Schindler y W. Killmann, Evaluation Criteria for True (Physical) Random Number Generators Used in Cryptographic Applications, Heildeberg: Springer, 2003.

[10] E. H. Y.-S. K. B. K. Ihor Vasyltsov, Fast Digital TRNG Based on Metastable Ring Oscillator, 2008 ed., vol. pp. 164–180, 2008.

[11] «random.org,» [En línea]. Available: http://www.random.org/randomness/. [Último acceso: 2013].

[12] B. Sunar, W. J. Martin y D. R. Stinson, A Provably Secure True Random Number Generator with Built-in Tolerance to Active Attacks, Worcester.

[13] P. Kohlbrenner y M. Lockheed, An Embedded True Random Number Generator

for FPGAs, Fairfax, USA.

[14] P. Ralzemond D., «Telegraph ciphering key tape machine». Estados Unidos de América Patente US 2406031 A, 20 Agosto 1946.

[15] J. Mechalas, «Intel,» Septiembre 2013. [En línea]. Available: http://software.intel.com/en-us/blogs/2012/11/17/the-difference-between-rdrand-and-rdseed.

[16] R. A. Schulz, «Random Number Generator Circuit». United States of America Patente 4905176, 27 February 1990.

[17] M. Vratislav, On the Low-power Design, Stability Improvement and Frequency Estimation of the CMOS Ring Oscillator, Grenoble.

[18] M. S. G. Jovanovic, A method for improvement stability of a CMOS voltage controlled ring oscillators, 2007.

[19] V. J, Practical random number generation in software, Virgina Tech, USA: Computer Security Applications Conference, 2003.

[20] P. Chambers, The Ten Commandments of Excellent, 1997.

[21] Randy, Sequential Logic Design, 1996.

[22] J. Stephenson y D. Chen, Understanding Metastability in FPGAs, San Jose, USA: Altera, 2009.

[23] P. Glover y E. Steve, Relationally Placed Macros, Xilinx, 2008.

[24] G. Marsaglia, A Bettery of tests of randomness.

[25] Xilinx, Constraints Guide, Xilinx Co., 2008.

[26] «Lavarnd,» [En línea]. Available: http://www.lavarnd.org/. [Último acceso: 2013].

[27] P. Horowitz y W. Hill, The Art of Electronics, Cambridge University Press, 1980.

[28] C. Boutin, «Pi seems a good random number generator – but not always the best,» Purdue University, 2005.

[29] C. Moler, Numerical Computing with MATLAB, Siam, 2004.

6. Presupuesto

Mano de obra 21.000 €

Concepto	Horas	Coste por horas	Total
Ingeniero Junior	300	50 € / h	15.000 €
Ingeniero Sénior	60	100 € / h	6.000 €

Infraestructura 12.000 €

Concepto	Meses	Coste por mes	Total
Alquiler de oficina	4	300 €	12.000 €

Software requerido 2.343,85 €

Concepto	Importe total	% amortización	Imputado
Licencia Microsoft Windows7	279,99 €	10 %	27,99
Licencia Office	99 €	10 %	9,99 €
ISE Design Suite	2.701,58 € *	50 %	1.350,79 €
Modelsim	710,15 € *	50 %	355,08 €
Matlab	2400 €	25 %	600 €

Hardware 605 €

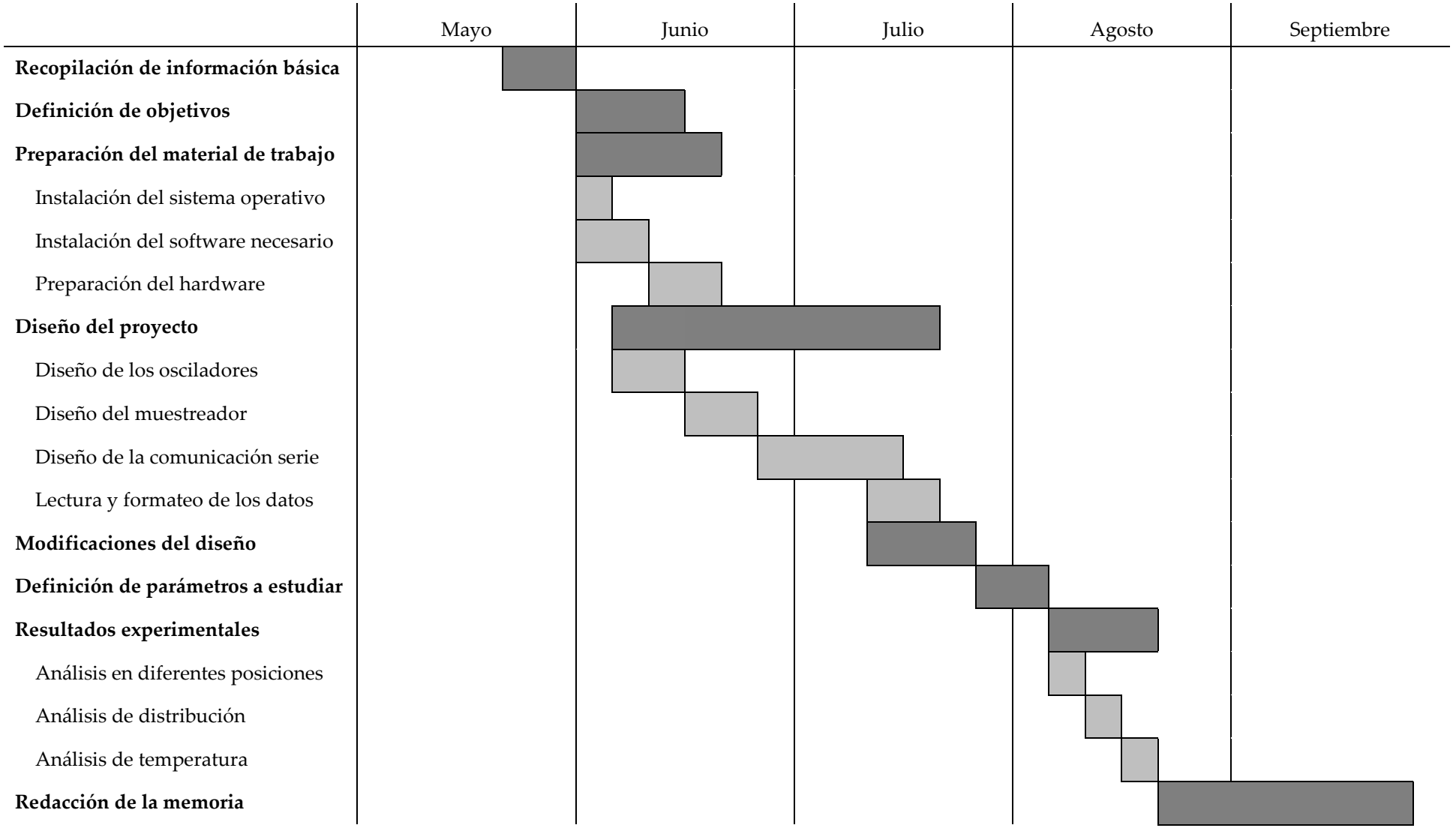
Concepto	Importe total	% amortización	Imputado
PC Intel Core2Duo 2.26GHz	600 €	5 %	30 €
FPGA Spartan 3E	200 €	50 %	100 €
Cable serie de tipo directo	10 €	100 %	10 €
PCMCIA puerto serie	15 €	100 %	15 €
Osciloscopio marca HAMEG	1.800 €	25 %	450 €

Resumen 28.948,85 €

Sección	Total
Mano de obra	21.000 €
Infraestructura	12.000 €
Software	2.343,85 €
Hardware	605 €
Total	**35.948,85 €**

*Se ha usado el cambio $1/0.75 € para las conversiones de divisa

7. Fases del proyecto



8. Normativa

En la legislación española, la generación de números aleatorios queda regulada por el Real Decreto 1613/2011, de 14 de noviembre, "por el que se desarrolla la Ley 13/2011, de 27 de mayo, regulación del juego, en lo relativo a los requisitos técnicos de las actividades de juego". En particular, los artículos que regulan los generadores de números aleatorios son:

Artículo 5. Requisitos del generador de números aleatorios:

El generador de números aleatorios, sin perjuicio de los requisitos adicionales que la Comisión Nacional del Juego pudiera establecer, deberá reunir, al menos, las siguientes características:

a) Los datos aleatorios generados serán imprevisibles e indeterminables.

b) Las series de datos generados no serán reproducibles.

c) Los métodos de escalamiento serán lineales y no introducirán ningún sesgo, patrón o predictibilidad.

d) El método de translación de los símbolos o resultados del juego no estará sometido a la influencia o control de un factor distinto de los valores numéricos derivados del generador de números aleatorios

Artículo 8. Plazo y procedimiento de homologación y certificación:

En relación con el generador de números aleatorios, el informe indicará el nivel de calidad intrínseca del generador, tras superar cuantas pruebas estadísticas sean necesarias para demostrar que los datos generados son de carácter aleatorio, imprevisibles, no reproducibles, y que los métodos de escalamiento y translación son lineales e independientes de cualquier otro factor que no sea el propio generador.

9. Anexo

9.1. Código fuente

A continuación se adjunta, a modo de apéndice el código necesario para la implementación del proyecto. La estructura de la interdependencia de los ficheros es la siguiente:

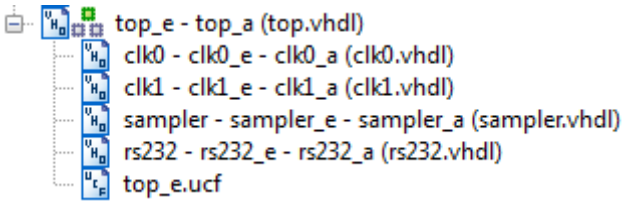


Ilustración 43 - Interdependencia de los ficheros del proyecto

Donde:

- top.vhdl es el fichero que une todos los demás ficheros
- clk0.vhdl es el oscilador número 0
- clk1.vhdl es el oscilador número 1
- sampler.vhdl es el muestreador
- rs232.vhdl es el módulo de comunicaciones serie
- top_e.ucf es el fichero de restricciones

9.1.1. Código VHDL

*9.1.1.1. Oscilador clk0_s**

```
entity clk0_e is
  port(
    Enable_i      : in std_logic;
    ClkOut_o      : out std_logic
    );
end clk0_e;

-----------------------------------------------

architecture clk0_a of clk0_e is
  signal a_s :  std_logic := '0';
  signal b_s :  std_logic := '0';
  signal c_s :  std_logic := '0';
  signal d_s :  std_logic := '0';
  signal e_s :  std_logic := '0';
  signal f_s :  std_logic := '0';
  signal g_s :  std_logic := '0';
  signal h_s :  std_logic := '0';


attribute keep: boolean;
attribute keep of a_s: signal is true;
attribute keep of b_s: signal is true;
attribute keep of c_s: signal is true;
attribute keep of d_s: signal is true;
attribute keep of e_s: signal is true;
attribute keep of f_s: signal is true;
attribute keep of g_s: signal is true;
attribute keep of h_s: signal is true;

begin
  b_s <= Enable_i and(a_s);
  c_s <= not(b_s);
  d_s <= not(c_s);
  e_s <= not(d_s);
  f_s <= not(e_s);
  g_s <= not(f_s);
  h_s <= not(g_s);
  a_s <= not(h_s);

  ClkOut_o <= b_s;
end clk0_a;
```

*NOTA: para la implentación de clk1_s se recomienda copiar este fichero y sustituir todas las cadenas clk0_s por clk1_s.

9.1.1.2. Muestreador

```
-------------------------------------------
-- Circuit which samples a clock signal
-- input using another.
-- In our case the frequencies of both
-- signals should be very similar.
-- Therefore the input comes from the
-- ring counters
-------------------------------------------

library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.all;

-------------------------------------------

entity sampler_e is
  port(
    Clk0_i     : in std_logic;  -- From ring oscillators 0 and 1
    Clk1_i     : in std_logic;
    CP_i       : in std_logic;
    reset_i    : in std_logic;
    preset_i   : in std_logic_vector (7 downto 0);
    S0_o       : out std_logic;

    ReadAck_i  : in std_logic;  -- Init for TR FF. Asynch.
    RandOut_o  : out std_logic; -- Radom BIT
    BitReady_o : out std_logic  -- Do we have a random bit ready?
    );
end sampler_e;

-------------------------------------------

architecture sampler_a of sampler_e is

  signal S0_s   : std_logic := '0';
  signal C0_s   : std_logic := '0';
  signal R0_s   : std_logic := '0';
  signal wait_s  : std_logic_vector (7 downto 0);
  signal clear_s : std_logic := '0';
  signal hold_s : std_logic := '0';
  signal rstHold_s : std_logic := '0';
  signal cnt_s : std_logic_vector (7 downto 0) := "00000000";
  signal E0_s : std_logic := '0';
  signal ctrlclks_s : std_logic := '0';
  signal R0cnt_s : std_logic := '0';
```

```
begin
  S0_o <= S0_s;
  --preset_s <= "00001000";
  ctrlclks_s <= CP_i;


  process(Clk0_i, Clk1_i, R0_s, ReadAck_i, S0_s)
  begin
    -- Each block represents a different flip flop
    if (Clk1_i'event and Clk1_i='1') then -- Top-Left FF
      S0_s <= Clk0_i;
    end if;

    if (R0_s = '1') then    -- Down-Left FF
      C0_s <=  '0';
    elsif (Clk1_i'event and Clk1_i= '0') then
      C0_s <= not(C0_s);
    end if;

    if (ReadAck_i = '1') then -- Top-Right FF
      BitReady_o <= '0';
    elsif (S0_s'event and S0_s='1') then
      if (E0_s = '0') then -- CE low-active
        BitReady_o <= '1';
      end if;
    end if;

    if (S0_s'event and S0_s= '1') then  -- Down-Right FF
      if (E0_s = '0') then -- CE low-active
        RandOut_o <= C0_s;
      end if;
    end if;
  end process;

-------------------------------------------------------


---------------------- This is the control part

-- OPTION 1
--E0_s <= '0';
--R0_s <= '0';


----OPTION 2
process(reset_i, ctrlclks_s, rstHold_s, S0_s)
begin
```

```
    if (reset_i = '0') then
      hold_s <= '0';
    elsif (rstHold_s = '1') then
      hold_s <= '0';
    elsif (S0_s'event and S0_s = '1') then
      if (hold_s <= '0') then
          hold_s <= '1';
       end if;
    end if;

    if (reset_i = '0') then
      cnt_s <= "00000000";
      R0cnt_s <= '0';
      E0_s <= '0';
      rstHold_s <= '1';
    elsif (ctrlclks_s'event and ctrlclks_s = '1') then
       if (hold_s = '1') then
            if (R0cnt_s = '0') then
                R0_s <= '1';
                E0_s <= '1';
                R0cnt_s <= '1';
            else
                R0_s <= '0';
                E0_s <= '1';
            end if;

            if (S0_s = '0') then
                if (cnt_s = PRESET_i) then
                    cnt_s <= "00000000";
                    rstHold_s <= '1';
                    E0_s <= '0';
                else
                    cnt_s <= cnt_s + 1;
                    E0_s <= '1';
                end if;
            end if;

       elsif (hold_s = '0') then
         rstHold_s <= '0';
         cnt_s <= "00000000";
         R0cnt_s <= '0';
         E0_s <= '0';
         R0_s <= '0';
        end if;
    end if;
end process;

end sampler_a;
```

9.1.1.3. RS-232

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.all;
-------------------------------------------
entity rs232_e is
  port(
   Rand_i           : in std_logic;
   BitReady_i       : in std_logic;
    CP_i                : in std_logic;
    Reset_n_i       : in std_logic;

    SerialOut_o     : out std_logic; -- To control unit
   ReadAck_o        : out std_logic -- Radom BIT
    );
end rs232_e;
-------------------------------------------
architecture rs232_a of rs232_e is
    signal selreg_s    : std_logic_vector(3 downto 0) := "0000";
    signal reg_s       : std_logic_vector (7 downto 0) := "00000000";
    signal selbitout_s : std_logic_vector (3 downto 0) := "0000";
    signal baudclk_s   : std_logic := '0';
    signal freqdiv_s   : std_logic_vector (8 downto 0):= "000000000";
    signal rxOtx_s     : std_logic := '0';   --Recieving data from trng (0) or
transmiting to PC (1)
    signal bitreadybuf0_s : std_logic := '0';
    signal bitreadybuf1_s : std_logic := '0';
    signal reset_bitreadybuf_s : std_logic := '0';
begin
process(CP_i, Reset_n_i)
  begin
    if (Reset_n_i = '0') then
        selreg_s <= "0000";
        bitreadybuf0_s <= '0'; --reset buffers
        bitreadybuf1_s <= '0';
        ReadAck_o <= '0';
        selbitout_s <= "0000";
        freqdiv_s <= "000000000";
        rxOtx_s <= '0';
        reg_s <= "00000000";
        SerialOut_o <= '1';
    elsif (CP_i'event and CP_i = '1') then
        if (rxOtx_s = '1') then     -- RS232 transmitting to PC
          bitreadybuf0_s <= '0';
          bitreadybuf1_s <= '0';
          ReadAck_o <= '0';
```

```
  selreg_s <= "0000";       -- reset selreg when not in use
  if (freqdiv_s = "110110010") then
     freqdiv_s <= "000000000";
    baudclk_s <= '1';
  elsif (freqdiv_s = "000000000") then
     freqdiv_s <= freqdiv_s + 1;
     baudclk_s <= '0';
     selbitout_s <= selbitout_s + 1;
     case selbitout_s is
         when "0000" => serialOut_o <= '0';
         when "0001" => serialOut_o <= reg_s(0);
         when "0010" => serialOut_o <= reg_s(1);
         when "0011" => serialOut_o <= reg_s(2);
         when "0100" => serialOut_o <= reg_s(3);
         when "0101" => serialOut_o <= reg_s(4);
         when "0110" => serialOut_o <= reg_s(5);
         when "0111" => serialOut_o <= reg_s(6);
         when "1000" => serialOut_o <= reg_s(7);
         when "1001" => serialOut_o <= '1';
                                selbitout_s <= "0000";
                                rxOtx_s <= '0';
         when others => serialOut_o <= '1';
                                selbitout_s <= "0000";
                                rxOtx_s <= '0';
     end case;
    else
     freqdiv_s <= freqdiv_s + 1;
     baudclk_s <= '0';
    end if;
else
  Serialout_o <= '1';
if (BitReadybuf1_s = '1') then
  bitreadybuf0_s <= '0';
     bitreadybuf1_s <= '0';
     ReadAck_o <= '1';
     selreg_s <= selreg_s +1;    -- only one increment line needed
     case selreg_s is
         when  "0000"  =>  reg_s(0) <= rand_i;
         when  "0001"  =>  reg_s(1) <= rand_i;
         when  "0010"  =>  reg_s(2) <= rand_i;
         when  "0011"  =>  reg_s(3) <= rand_i;
         when  "0100"  =>  reg_s(4) <= rand_i;
         when  "0101"  =>  reg_s(5) <= rand_i;
         when  "0110"  =>  reg_s(6) <= rand_i;
         when  "0111"  =>  reg_s(7) <= rand_i;
                                rxotx_s  <= '1';
         when others =>  selreg_s <= "0000";
        end case;
```

```
        else
            bitreadybuf0_s <= BitReady_i;
             bitreadybuf1_s <= bitreadybuf0_s;
          ReadAck_o <= '0';
        end if;
        end if;
    end if;
end process;
--signal bitneumann_s : std_logic := '0';                    -----Von neuman
reduction
--signal concat_s : std_logic_vector (1 downto 0);
--
-- begin
--process(CP_i, Reset_n_i)
--  variable var : std_logic;
--  variable cntneumann : std_logic;
--  variable store : std_logic;
--  begin
--      if (Reset_n_i = '0') then
--          selreg_s <= "0000";
--          bitreadybuf0_s <= '0'; --reset buffers
--      bitreadybuf1_s <= '0';
--      ReadAck_o <= '0';
--      selbitout_s <= "0000";
--      freqdiv_s <= "000000000";
--      rxOtx_s <= '0';
--      reg_s <= "00000000";
--      SerialOut_o <= '1';
--      elsif (CP_i'event and CP_i = '1') then
--          if (rxOtx_s = '1') then     -- RS232 transmitting to PC
--        bitreadybuf0_s <= '0';
--        bitreadybuf1_s <= '0';
--        ReadAck_o <= '0';
--        selreg_s <= "0000";       -- reset selreg when not in use
--            if (freqdiv_s = "110110010") then
--           freqdiv_s <= "000000000";
--              baudclk_s <= '1';
--        elsif (freqdiv_s = "000000000") then
--           freqdiv_s <= freqdiv_s + 1;
--           baudclk_s <= '0';
--           selbitout_s <= selbitout_s + 1;
--           case selbitout_s is
--               when "0000" => serialOut_o <= '0';
--               when "0001" => serialOut_o <= reg_s(0);
--               when "0010" => serialOut_o <= reg_s(1);
--               when "0011" => serialOut_o <= reg_s(2);
--               when "0100" => serialOut_o <= reg_s(3);
--               when "0101" => serialOut_o <= reg_s(4);
--               when "0110" => serialOut_o <= reg_s(5);
--               when "0111" => serialOut_o <= reg_s(6);
--               when "1000" => serialOut_o <= reg_s(7);
--               when "1001" => serialOut_o <= '1';
--                                  selbitout_s <= "0000";
--                                  rxOtx_s <= '0';
--               when others => serialOut_o <= '1';
--                                  selbitout_s <= "0000";
--                                  rxOtx_s <= '0';
--           end case;
--         else
```

```
--           freqdiv_s <= freqdiv_s + 1;
--           baudclk_s <= '0';
--          end if;
--      else  -- Recieve
--            Serialout_o <= '1';
--        if (BitReadybuf1_s = '1') then
--          bitreadybuf0_s <= '0';
--           bitreadybuf1_s <= '0';
--           ReadAck_o <= '1';
--              if (cntneumann = '0') then
--                  bitneumann_s <= rand_i;
--                  cntneumann := '1';
--              elsif (cntneumann = '1') then
--                  concat_s <= rand_i & bitneumann_s;
--                  case concat_s is
--                      when "01" => case selreg_s is
--                                       when  "0000"  =>  reg_s(0) <= '0';
--                                       when  "0001"  =>  reg_s(1) <= '0';
--                                       when  "0010"  =>  reg_s(2) <= '0';
--                                       when  "0011"  =>  reg_s(3) <= '0';
--                                       when  "0100"  =>  reg_s(4) <= '0';
--                                       when  "0101"  =>  reg_s(5) <= '0';
--                                       when  "0110"  =>  reg_s(6) <= '0';
--                                       when  "0111"  =>  reg_s(7) <= '0';
--                                                         rxotx_s  <=
'1';
--                                       when others =>  selreg_s <= "0000";
--                                   end case;
--                                selreg_s <= selreg_s +1;
--                      when "10" => case selreg_s is
--                                       when  "0000"  =>  reg_s(0) <= '1';
--                                       when  "0001"  =>  reg_s(1) <= '1';
--                                       when  "0010"  =>  reg_s(2) <= '1';
--                                       when  "0011"  =>  reg_s(3) <= '1';
--                                       when  "0100"  =>  reg_s(4) <= '1';
--                                       when  "0101"  =>  reg_s(5) <= '1';
--                                       when  "0110"  =>  reg_s(6) <= '1';
--                                       when  "0111"  =>  reg_s(7) <= '1';
--                                                         rxotx_s  <=
'1';
--                                       when others =>  selreg_s <= "0000";
--                                   end case;
--                                selreg_s <= selreg_s +1;
--                      when others => cntneumann := '0';
--                  end case;
--                  cntneumann := '0';
--              end if;
--
--          else
--          bitreadybuf0_s <= BitReady_i;
--           bitreadybuf1_s <= bitreadybuf0_s;
--          ReadAck_o <= '0';
--        end if;
--      end if;
--  end if;
--end process;
end rs232_a;
```

9.1.1.4. Top (todos los módulos juntos)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity top_e is
  port(
     CP_i        : in std_logic;
    ena_clk1_i   : in std_logic;
    ena_clk0_i   : in std_logic;
    --readack_i   : in std_logic;
     clk0_o : out std_logic;
     clk1_o : out std_logic;
    reset_i     : in std_logic;
    PRESET_i     : in std_logic_vector (7 downto 0);
     serialout_o : out std_logic;
     S0_o            : out std_logic

    );
end top_e;

architecture top_a of top_e is
  component clk0_e
    port(
    Enable_i       : in std_logic;
    ClkOut_o       : out std_logic
      );
  end component;

  component clk1_e
    port(
    Enable_i       : in std_logic;
    ClkOut_o       : out std_logic
      );
  end component;

  component sampler_e
    port(
    Clk0_i      : in std_logic;  -- From ring oscillators 0 and 1
    Clk1_i      : in std_logic;
    CP_i        : in std_logic;
    reset_i     : in std_logic;
    preset_i    : in std_logic_vector (7 downto 0);
    S0_o        : out std_logic;

    ReadAck_i   : in std_logic;  -- Init for TR FF. Asynch.
    RandOut_o   : out std_logic; -- Radom BIT
```

```
    BitReady_o : out std_logic  -- Do we have a random bit ready?
      );
  end component;

    component rs232_e
  port(
    Rand_i          : in std_logic;
    BitReady_i  : in std_logic;
     CP_i               : in std_logic;
     Reset_n_i      : in std_logic;

     SerialOut_o    : out std_logic; -- To control unit
    ReadAck_o   : out std_logic -- Radom BIT
    );
    end component;

  --All the signals are declared here,which are not a part of the top module.
  --
  signal all_sent_s : std_logic := '0';  -- done sending all data
  signal clk0_s : std_logic := '0';
  signal clk1_s : std_logic := '0';
  signal S0_s   : std_logic := '0';
  signal E0_s   : std_logic := '0';
  signal R0_s   : std_logic := '0';

  signal randout_s  : std_logic := '0';
  signal bitready_s : std_logic := '0';
  signal readack_s : std_logic := '0';

begin
    clk1_o <= clk1_s;
    clk0_o <= clk0_s;

  clk0 : clk0_e port map (
    Enable_i => ena_clk0_i,
    clkout_o => clk0_s
    );

  clk1 : clk1_e port map (
    Enable_i => ena_clk1_i,
    clkout_o => clk1_s
    );


  sampler : sampler_e port map(
    Clk0_i     => clk0_s,
    Clk1_i     => clk1_s,
    CP_i      => CP_i,
```

```
    reset_i   => reset_i,
    preset_i  => "00000100",
    S0_o      => S0_o,
    readack_i => readack_s,
    randout_o => randout_s,
    bitready_o => bitready_s
    );

    rs232 : rs232_e port map(
    Rand_i          => RandOut_s,
    BitReady_i  => bitready_s,
     CP_i               => CP_i,
     Reset_n_i      => reset_i,

     SerialOut_o    =>  SerialOut_o,
    ReadAck_o   => readAcK_s
     );

end top_a;
```

9.1.1.6. Fichero de restricciones (.udf)

```
INST "clk0/a_s1" U_SET=myrpm;
INST "clk0/b_s1" U_SET=myrpm;
INST "clk0/c_s1" U_SET=myrpm;
INST "clk0/d_s1" U_SET=myrpm;
INST "clk0/e_s1" U_SET=myrpm;
INST "clk0/f_s1" U_SET=myrpm;

INST "clk1/a_s1" U_SET=myrpm;
INST "clk1/b_s1" U_SET=myrpm;
INST "clk1/c_s1" U_SET=myrpm;
INST "clk1/d_s1" U_SET=myrpm;
INST "clk1/e_s1" U_SET=myrpm;
INST "clk1/f_s1" U_SET=myrpm;

INST "sampler/S0_s" U_SET=myrpm;
INST "sampler/C0_s" U_SET=myrpm;

INST "clk0/a_s1" RLOC = X0Y0;
INST "clk0/b_s1" RLOC = X0Y1;
INST "clk0/c_s1" RLOC = X0Y2;
INST "clk0/d_s1" RLOC = X0Y3;
INST "clk0/e_s1" RLOC = X0Y4;
INST "clk0/f_s1" RLOC = X0Y5;
INST "clk0/g_s1" RLOC = X0Y6;
INST "clk0/h_s1" RLOC = X0Y7;

INST "clk1/a_s1" RLOC = X2Y0;
INST "clk1/b_s1" RLOC = X2Y1;
INST "clk1/c_s1" RLOC = X2Y2;
INST "clk1/d_s1" RLOC = X2Y3;
INST "clk1/e_s1" RLOC = X2Y4;
INST "clk1/f_s1" RLOC = X2Y5;
INST "clk1/g_s1" RLOC = X2Y6;
INST "clk1/h_s1" RLOC = X2Y7;

INST "sampler/S0_s" RLOC = X1Y0;
INST "sampler/C0_s" RLOC = X1Y1;

INST "clk0/a_s1" RLOC_ORIGIN = X0Y0;


NET "clk1_o" LOC = C7;
NET "CP_i"       LOC = C9;
NET "ena_clk1_i" LOC = H18;
```

```
NET "ena_clk0_i" LOC = N17;
NET "reset_i"    LOC = L13;
NET "serialout_o" LOC = "M14" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW = SLOW;
NET "S0_o"          LOC = D7;
```

9.1.2. Banco de pruebas de los bloques RS-232 y muestreador

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;

-- Entity declaration for your testbench.
-- Dont declare any ports here.
ENTITY samNrs_tb IS
END samNrs_tb;

ARCHITECTURE behavior_a OF samNrs_tb IS
    COMPONENT test
    port(
     CP_i       : in std_logic;
    clk1_i   : in std_logic;
    clk0_i   : in std_logic;

    reset_i    : in std_logic;
    PRESET_i    : in std_logic_vector (7 downto 0);
     serialout_o : out std_logic;
     S0_o           : out std_logic
    );
    END COMPONENT;

    for all : test use entity work.snr_e (snr_a);

    signal Clk0_i     : std_logic := '0';
    signal Clk1_i     : std_logic := '0';
    signal CP_i       : std_logic := '0';
    signal reset_i    : std_logic := '0';
    signal preset_i   : std_logic_vector (7 downto 0) := "00000000";
    signal S0_o       : std_logic := '0';

    ----------------------------------------------------
    -- Clock period definitions
    -- Freq of Clk0_i and Clk1_i should be similar:
    constant clk_period_0 : time := 10.01 ns;
    constant clk_period_1 : time := 10.04 ns;
    constant CP_period    : time := 20 ns;
    ----------------------------------------------------

BEGIN
  -- Instantiate the Unit Under Test (UUT)
   uut: test PORT MAP (
    Clk0_i => clk0_i,
    Clk1_i => clk1_i,
    CP_i   => CP_i,
    reset_i => reset_i,
    preset_i => preset_i,
    S0_o     => S0_o
      );

    -- Clock process definitions (clock with 50% duty cycle is generated here)
   clk_0_process : process
   begin
        clk0_i <= '0';
        wait for clk_period_0/2;
        clk0_i <= '1';
```

```
        wait for clk_period_0/2;
    end process;

    clk_1_process : process
    begin
        clk1_i <= '0';
        wait for clk_period_1/2;
        clk1_i <= '1';
        wait for clk_period_1/2;
    end process;

   CP_process : process
    begin
        CP_i <= '0';
        wait for CP_period/2;
        CP_i <= '1';
        wait for CP_period/2;
    end process;

      -- Stimulus process
   stim_proc: process
    begin
     reset_i <= '0';
     wait for 1000 ns;
     reset_i <= '1';
     wait for 100000000 ns;
end process;
END;
```

9.1.3. Código Python

El siguiente código genera un fichero llamado "workfile.txt" en el directorio desde el que es ejecutado. Deberá guardarse con una extensión .py para poder ejecutarlo con doble click.

```
import binascii
import serial
import winsound
import time

if __name__ == "__main__":
    try:
        ser = serial.Serial()
        ser.baudrate = 115200
        ser.port = 'COM4'
        ser.timeout=5
        ser.open()
        f = open('workfile.txt', 'a')
        start_time = time.time()
        if ser.isOpen():
            bits = 100 #total number of desired bits in MILLIONS. At least 80.
            iterations = int((bits*1000000)/320)
            print (iterations)
            for i in range(iterations): #100 million bits (at least 80
                for j in range(40): # 10 integers of 8 hex digits each per line->
320 bits
                    f.write(str(binascii.hexlify(ser.read(1)))[2:4])
                f.write("\n")
                #print (str(i)+' of '+str(iterations))
                print (repr(i).rjust(2), repr("of").rjust(3), repr
(iterations).rjust(4))
            print("**********DONE**********")
            winsound.Beep(440, 5000)
        else:
            print ("Serial port in COM4 is not OPEN")
        elapsed_time = time.time() - start_time
        print(elapsed_time)
        print("Press ENTER")
        f.close()
        input()
    except serial.serialutil.SerialException:
            print ("EXCEPT0: COM4 NOT DETECTED")
```